ChatGPT

システム設計書: ローカル環境AIチャットツール

1. システム概要

背景・課題: 現在、社内の「過去の懸案事項と対策内容」はExcelや個別のドキュメントで管理されています。しかしExcelでのナレッジ共有には**複数人での編集が難しい・メンテナンスが煩雑**など多くのデメリットがあり [1] [2]、対応策を過去に記録していても**適切に検索・参照できず忘れてしまう**問題が発生しています。また社内情報をクラウドのAIに投入するのは**機密漏洩やコンプライアンス上の懸念**があります [3]。これらの背景から、**ローカル環境で完結するAIチャットツール**によってナレッジを活用しやすくすることが求められています。

目的・価値: 本システムは、社内で蓄積された過去の懸案とその対策（ナレッジ）をデータベース化し、ユーザーがチャット形式で質問するとAIが**登録データを参照して回答**できるようにするものです。これにより、担当者が過去の対応策を迅速に検索でき**業務の効率化**や**対応漏れの防止**につながります。またナレッジが個人に属人化せず組織で共有できるため、新任者の学習や問い合わせ対応の品質向上にも寄与します。

対象ユーザー:　一般ユーザー（社内の問い合わせ対応者など）はチャットで質問し回答を得たり、ナレッジデータを検索・閲覧できます。管理者ユーザーは懸案データの一括取り込みやシステム設定（LLMモデル変更等）を行います。
※ユーザー管理（ログイン）機能は必要に応じて導入しますが、ローカルネットワーク内のみで利用し権限管理がシンプルであれば省略も検討します（設計上の仮定）。

スコープ（やること）: 懸案データの登録（手入力およびファイル取り込み）、編集・削除、画像添付、懸案一覧・検索、AIチャットによるQ&A、チャット履歴の保存・閲覧、基本的な管理設定機能を含みます。社内完結型のためクラウドサービスとの連携や外部APIは使用しません。
スコープ（やらないこと）: 本システムは**社内ナレッジに基づくQA**に特化しており、インターネット上の一般知識検索や高度な分析（BI的分析機能）は行いません。また、画像からの自動内容理解（マルチモーダルな画像質問応答）など高度な機能は現時点では扱いません（将来拡張の余地として言及）。ユーザー管理も必要最低限に留め、細かな権限ロール設計などは範囲外とします。

前提・制約: 本システムは**ローカル環境のみで完結**し、Windows OS上で稼働することを前提とします。UIはWebブラウザ上で動作するシングルページアプリケーション（SPA）として、初心者でも扱いやすい形で実装します。外部のLLMサービスAPIは使用せず、Ollama等を利用した**ローカルLLMモデル**で回答生成します。同時接続ユーザーは最大100人程度、懸案データは約1000件規模を想定します。初心者が実装・運用できるよう構成を極力シンプルにし、複雑な分散システムではなく**単一サーバー上で完結**するアーキテクチャを採用します（設計上の判断）。LLM推論に時間がかかる可能性があるため、**リアルタイム性よりも正確さとプライバシー**を重視し、必要に応じてキューイングなどで同時要求に対応します。

2. アーキテクチャ設計

全体構成: 本システムはフロントエンド（ブラウザSPA）、バックエンド（アプリケーションサーバ）、データベース、ローカルLLM推論モジュール、ファイルストレージから構成されます。フロントエンドはReactベースのSPAで、ユーザーと対話(UI表示や入力)を担います。バックエンドはNode.jsまたはPythonで実装されるWebサーバ（後述の技術選定で詳細）で、RESTfulなHTTP　APIを提供しフロントエンドと通信します。データベースは**SQLite**を使用し、懸案データやチャット履歴、添付ファイルメタ情報などを格納します。添付画像ファイルやインポートしたドキュメントは**サーバーのローカルファイルストレージ**上に保存し、DBに

はパス等のメタデータのみ保持します。LLMはOllama等によりローカルでホストされたモデルを使用し、バックエンドは内部的にLLMへ問い合わせを行います。通信方式としては、フロントエンドとバックエンド間はHTTP(S)+JSONで通信し、バックエンドからLLMモジュールへは**ローカルのHTTP API**（OllamaのREST API等）やコマンド実行を通じてプロンプトを送信し回答を取得します [4]。全ての処理が社内LANもしくはスタンドアロンPC内で完結し、インターネットに依存しない構成です。

構成図: 以下に全体のコンポーネント構成を示します（Mermaid記法の図）。

```
graph LR
  subgraph Frontend（ブラウザSPA）
    UI[Reactアプリ（ブラウザ）]
  end
  subgraph Backend（アプリサーバ）
    API[Web APIサーバ<br/>(Node.js / FastAPI)]
    DB[(SQLite<br/>データベース)]
    Files[ファイルストレージ<br/>(添付画像・インポート)]
    LLM[ローカルLLM推論<br/>(Ollamaサーバ)]
  end
  UI -- HTTP/JSON --> API
  API -- SQL（クエリ/更新）--> DB
  API -- ファイル読み書き --> Files
  API -- プロンプト呼び出し --> LLM
```

上図のように、フロントエンドはユーザー操作を受け付け、バックエンドのAPIにリクエストを送ります。バックエンドはDBとのデータ操作やファイル保存、LLMへのプロンプト送信などを行い、その結果をフロントへJSONで返します。LLM部分はOllamaなどが提供するローカルAPIを経由してモデル推論を実行する想定です。なお、LLMモデル自体のデータ（数GBのモデルファイル）は事前にサーバ内に配置され、API経由でリクエストごとに対話（推論）を行います。

主要シーケンス図: 各主要機能におけるシステム内の処理フローをシーケンス図で示します。

- **(1) 懸案登録（手入力）**: ユーザーがWebフォームから懸案事項を手動登録する際の処理。

```
sequenceDiagram
    participant User as ユーザー
    participant Browser as ブラウザSPA
    participant Server as バックエンドAPI
    participant DB as SQLite DB

    User->>Browser: 懸案フォーム入力（タイトル/内容/対策 等）
    Browser->>Server: POST /issues （入力データ）
    Server->>DB: 懸案データINSERT
    DB-->>Server: 保存成功
    Server-->>Browser: 成功レスポンス（新規懸案ID等）
    Browser-->>User: 登録完了メッセージ表示
```

- **(2) 懸案登録（ファイル取り込み）**: 既存のWord/Excel/PDFファイルをアップロードして懸案を登録する処理。

```
sequenceDiagram
    participant User as ユーザー
    participant Browser as ブラウザSPA
    participant Server as バックエンドAPI
    participant FileParser as ファイル解析モジュール
    participant DB as SQLite DB

    User->>Browser: .docx/.xlsx/.pdfファイル選択・アップロード
    Browser->>Server: POST /issues/import （ファイルバイナリ）
    Server->>FileParser: アップロードファイル解析（テキスト抽出）
    FileParser-->>Server: 構造化された懸案データ案
    Server-->>Browser: 抽出データのプレビュー送信
    User->>Browser: プレビュー内容を確認・修正
    Browser->>Server: POST /issues/import/confirm （確定データ）
    Server->>DB: 懸案データINSERT（複数件の場合ループ）
    DB-->>Server: 保存結果OK
    Server-->>Browser: インポート結果（件数等）応答
    Browser-->>User: インポート完了メッセージ／結果表示
```

- **(3) 懸案への画像添付**: 懸案事項に関連するスクリーンショット等の画像ファイルをアップロード・保存する処理。

```
sequenceDiagram
    participant User as ユーザー
    participant Browser as ブラウザSPA
    participant Server as バックエンドAPI
    participant Storage as ファイルストレージ
    participant DB as SQLite DB

    User->>Browser: 画像ファイル選択（複数可）
    Browser->>Server: POST /issues/{id}/attachments （画像バイナリ）
    Server->>Storage: 画像ファイル保存（サーバ内ディレクトリ）
    Server->>DB: 添付メタ情報INSERT（パス等）
    DB-->>Server: 保存OK
    Server-->>Browser: アップロード結果（添付ID・URL等）
    Browser-->>User: 画像プレビュー表示（サムネイル等）
```

- **(4) チャット質問→RAG→回答生成→履歴保存**: ユーザーの質問に対し、関連懸案を検索してLLMが回答を生成し、履歴を保存する処理（Retrieval-Augmented Generationの流れ）。

```
sequenceDiagram
    participant User as ユーザー
    participant Browser as ブラウザSPA
    participant Server as バックエンドAPI
    participant DB as SQLite DB
    participant LLM as ローカルLLM

    User->>Browser: 質問を入力・送信
    Browser->>Server: POST /chat/question （質問テキスト）
```

```
Server->>DB: 懸案検索（全文検索またはベクトル検索）
DB-->>Server: 該当懸案データ 上位N件
Server->>LLM: プロンプト生成 & LLM問い合わせ（質問+関連データ）
LLM-->>Server: 回答テキスト生成
Server->>DB: チャット履歴INSERT（質問・回答内容、参照懸案ID等）
DB-->>Server: 保存OK
Server-->>Browser: 回答結果 （テキスト＋引用情報）
Browser-->>User: 回答をチャット画面に表示
```

上記フローにおいて、RAG（Retrieval-Augmented Generation）の部分ではバックエンドがDBから関連情報を取得してLLMに与えています。そのため回答には常に最新の懸案ナレッジが反映され、ユーザーからの質問に対し**高い関連性と正確性**を確保します [5] 。

同時100ユーザー時の処理方針:
本システムでは同時に最大100ユーザーからのリクエストを想定しています。バックエンドAPI自体はリクエストを非同期処理することで同時接続を捌けますが、課題となるのは**LLM推論処理の重さ**です。LLMは基本的に1リクエストずつ順番に処理する必要があるため、同時多数の質問要求が来た場合は内部で**リクエストのキューイング**を行い、LLM処理を直列化する設計とします（設計上の判断）。例えば、バックエンドでLLM呼び出し用のジョブキューを持ち、リクエストごとにキューに投入→ワーカーが順次処理する方式です。こうすることでサーバが高負荷でメモリ圧迫・クラッシュするのを防ぎます。応答待ちのユーザーにはUI上で「回答を生成中...」等のインジケータを表示し、一定時間（例えば20秒）以上応答が無い場合はタイムアウトエラーを返すようにします（上限時間も管理者設定可能とする）。

また**ストリーミング応答**の採用も検討します。LLMが文章を逐次生成できる場合、生成済み部分から順次フロントに送信すればユーザーは早めに内容を確認できます。バックエンドからフロントへServer-Sent Events（SSE）やWebSocket等でストリーム送信する方法もありますが、初心者実装の難易度を考慮し、まずは**一括応答**で実装し、後からストリーミングをオプションで追加する設計とします。

100ユーザー同時利用時には、DBアクセスやファイルI/Oも増えますがSQLiteは読取並列は比較的強く、書込はロック制御されます。同時書込要求が多数ある場合は、リトライ処理やキューで調整し、また検索系クエリにはインデックスを適切に付与して高速化します。さらに、将来的にユーザー数やデータ量が大きく増加する場合には、データベースをSQLiteからよりスケーラブルなRDBMS(PostgreSQL等)へ移行する余地を残しておきます（非機能要件で詳細記載）。

3. 機能一覧・要件詳細

3.1 ユーザー機能

- **懸案管理:** ユーザーは過去の懸案事項（問題と対策）をシステムに登録・編集・削除できます。主な項目は「タイトル（件名）」「詳細説明（問題の状況）」「対策内容（行った対応策）」「登録日」「登録者」などです。懸案の一覧表示・検索機能も提供し、キーワードによる全文検索で必要な懸案を絞り込めます（タグ分類の実装は後述）。検索UIにはフリーテキスト検索ボックスを設け、タイトル・内容・対策の全文を横断検索します。**タグ検索:** もし懸案にタグ付けしたい要望があれば、タグ（キーワード）を複数設定し、タグ一覧やタグによる絞り込みも可能です。ただしまずはシンプルな全文検索で実装し、タグ機能は**将来的な拡張**として検討します（初心者でもまず完成させやすい範囲を優先）。

- **懸案登録（手入力フォーム）:** Webフォームから新たな懸案を追加できます。フォーム項目は上記のタイトル・詳細・対策（複数行テキスト入力）等で、必要ならカテゴリやタグ入力欄も設けます。入力

内容をPOSTするとバックエンドがバリデーションを行い、問題なければDBに保存します。登録後、画面に「登録しました」と成功メッセージを表示し、登録内容を確認できる詳細画面に遷移します。

- **懸案編集・削除:** 既存の懸案詳細画面から編集操作が可能です。編集権限は原則登録者および管理者に限定し、内容を変更して保存できます。削除も同様に確認ダイアログのうえ実行し、DBからレコードおよび関連する添付ファイルを削除します。削除操作も管理者のみ許可するか、または削除ではなく「無効化フラグ」を立てて非表示にする等の運用も検討します（重要データの誤削除防止）。

- **懸案一覧・検索:**　全懸案を一覧表示する画面では、新しい順やタイトル順でソート表示し、ページングを実装します（例えば20件ずつ表示など）。検索はタイトル・内容に対するキーワード検索を提供し、SQLiteのFTS5仮想テーブルを用いた全文検索で効率的に照合します [6]。検索キーワードはAND/ORや部分一致にも対応させ、初心者実装の場合はシンプルにスペース区切りAND検索程度から開始します。

- **懸案ファイル取り込み（インポート）:**　既存のWord（.docx）やExcel（.xlsx）、PDF（.pdf）のファイルから懸案データを取り込む機能です。ユーザーはファイルをアップロードすると、サーバ側で**テキスト抽出→構造化→プレビュー表示**が行われます。例えば、Wordファイル中に複数の懸案が記載されている場合、それぞれを検出し、懸案タイトル・内容・対策に相当するテキストを抽出します。抽出結果は登録前に画面で確認・編集でき、ユーザーは誤認識や不要部分を修正してから確定登録します。複数件の懸案が一度に取り込まれる場合は、件数や各懸案のタイトルをプレビュー一覧で示し、選択したもののみ登録することも可能にします（例えば特定行だけインポートする等）。

- **懸案への画像添付:** 各懸案に対しスクリーンショット画像等を複数添付できます。懸案登録フォームや編集画面で「画像添付」ボタンを用意し、クリックするとファイル選択ダイアログが開きます。複数ファイル選択に対応し、一度に複数の画像をアップロード可能です。アップロード後はサーバ側で保存し、画面上でサムネイルプレビューを表示します。また各画像には**説明キャプション**を付ける入力欄も提供し、ユーザーが「エラー画面のスクショ」など簡単な説明を書けるようにします。このキャプションは後述のようにOCRテキストとともに検索対象に含めます。添付画像は後から削除も可能で、編集画面で不要な画像に対し「削除」操作を提供します（削除時はサーバからファイルも物理削除）。

- **チャット質問・回答取得:**　メインとなるチャットUI上で、ユーザーはテキストで質問を入力し、送信（Enterキーまたは送信ボタン）します。するとAI（LLM）が回答文を生成して画面に表示します。ユーザー側には自分の質問とAIからの回答が対話形式で時系列に表示され、過去のQAがスクロールで辿れる**履歴**となります。回答生成中はメッセージエリアに「AIが考えています...」などと表示し、生成完了後に回答メッセージに置き換えます。LLMの応答には、可能であれば**参照した懸案情報の出典**が含まれるように設計します（例：「...過去の懸案No.12によれば...」等）。ただしモデル出力に出典を埋め込む制御は難度が高いため、MVP段階ではシステム側で回答後に補足を付け足すか、ユーザーが手動で参照ボタンを押すと関連懸案を表示するなどUI上で工夫する案も検討します（設計上の代替案）。

- **チャット履歴閲覧・エクスポート:**　チャット画面ではその場で過去のやりとりをスクロール表示できますが、別途**履歴一覧**画面を用意し、過去のQ&A履歴をまとめて見ることも可能にします（ユーザー単位で自分の履歴、または管理者なら全ユーザーの履歴を参照）。また、任意機能として履歴をテキストファイル（例えばMarkdownやCSV）にエクスポートする機能も検討します。これにより重要なQAを別途まとめたり、ナレッジ更新の参考にしたりできます。
※エクスポート機能は必須ではないため、優先度低としてMVPでは実装せず、将来的な拡張と位置づけます。

- **懸案とチャット履歴の紐づけ:** チャットで生成した回答がどの懸案データを参照したか追跡できるようにします。例えば、回答が参照した懸案IDを内部的に記録しておき、後で「この懸案はチャット回答〇〇で使用された」といった関連が見られるようにします。UI上の例としては、チャット履歴の各回答メッセージに「参照懸案を見る」リンクを設置し、クリックするとその懸案詳細を開く、といった実装です。逆に懸案詳細画面で「この懸案が使われた回答一覧」を表示することも考えられます。これらの紐づけにより、AI回答の信頼性を裏付けたり、懸案データが本当に役立っているかを分析できるようになります。実装コストとの兼ね合いもあるため、MVP時点では**内部的な記録のみ**に留め、UIでの表示は管理者向け画面で簡易的に確認できる程度とするかもしれません（設計上の仮定）。

3.2 管理者機能

- **モデル切替設定:** 管理者は使用するLLMモデルを変更できます。具体的には、Ollama等で利用可能なモデルの名前やパラメータを設定画面で入力し、保存すると次回のチャット回答からそのモデルを使用します。例えば「現在のモデル: `Mistral-7B`」を「`Llama-2-13b`」に変更して適用、などが可能です。モデル変更後は念のためテスト質問で応答が正しく得られるか確認する運用を想定します。モデルの**導入（ダウンロード）**自体はシステム外で事前に行っておき（Ollamaコマンドでpullする等）、当設定では**モデル名やパスの指定**のみを扱う想定です。

- **システム設定:** LLMの動作パラメータやタイムアウト時間等、システム全体の設定値を管理者が調整できるようにします。例として、**最大トークン数**（1回答あたりのトークン上限）、**温度**（回答のランダム性）、**トップK**や**トップP**といった生成パラメータ、**回答待ちタイムアウト**秒数、**検索件数N**（RAGで何件の懸案を文脈提供するか）などです。設定画面で各値を入力・選択し、保存するとバックエンドの設定に反映されます。これらの設定値はSQLiteの設定テーブルに保存するか、設定ファイル(JSON/YAML)に書き出してサーバー再起動時に読み込む形にする予定です。初心者でも扱いやすいよう、UI上は適切なデフォルト値を提示し、説明文や入力範囲のバリデーションも付けます。

- **ユーザー管理:** システムを複数ユーザーで利用する場合、管理者はユーザーアカウントの登録・削除やパスワードリセット等を行える機能を持ちます。具体的には、新規ユーザーの追加（ユーザー名、初期パスワード、権限ロール）、既存ユーザー情報の変更（表示名、権限変更など）、不要アカウントの削除が可能です。もっとも、小規模社内ツールでユーザー数も限られる想定のため、ユーザー管理は**必須ではない**かもしれません。システム利用者全員が共通アカウントで使うケースやWindowsログオンユーザーと連携するケースも考えられます。設計上はユーザー管理機能を用意しますが、MVPでは**管理者1名＋一般ユーザー複数の固定構成**として、UIからユーザー管理を省略する判断もあり得ます（この場合ユーザーの識別はクッキーや端末IDで代用）。

3.3 取り込み（インポート）仕様（重要）

対象ファイル: `.docx`（Word）、`.xlsx`（Excel）、`.pdf`（PDF）の3種類をサポート対象とします。これらは社内で過去に作成された懸案報告書やナレッジ資料の形式を想定しています。なお、旧拡張子の`.doc`や`.xls`、`.pptx`などは対応しません（必要なら一度Officeで開いて保存し直す運用とします）。

テキスト抽出: アップロードされたファイルはサーバ側でまず**テキストを抽出**します。各形式ごとの処理は以下の通りです。

- Word（.docx）: XMLベースのOffice Open XML形式であるため、ライブラリを用いて文書内テキストをすべて抽出します。段落や見出し、表のセル内容も含めて取得可能です。Node.jsの場合、`officeparser`や`office-text-extractor`等のNPMライブラリでdocxからテキストを取り出せます[7]。Pythonでは`python-docx`や`textract`ライブラリで同様に読み取り可能です[8]。

- Excel（.xlsx）: スプレッドシート内のセルを読み取ります。懸案情報をExcelで管理している場合、通常は1行＝1懸案として各列に項目（タイトル列、詳細列、対策列等）があると考えられます。その前提で、1行ずつ読み込み各セル値を対応項目に割り当てます。Node.jsでは`xlsx`（SheetJS）ライブラリでExcelを読み込めますし、Pythonでは`openpyxl`や`pandas`でExcelシートを読み込めます。Excel内に画像や図表が埋め込まれている場合、それらの抽出はスコープ外としテキストデータのみ対象とします。

- PDF（.pdf）: PDFはレイアウト情報を含むフォーマットです。テキスト抽出にはPDFパーサ（例えばNodeの`pdf-parse`、Pythonの`PyMuPDF`/`fitz`等）を用います。これらでページごとのテキストを取得し、改ページの区切りも保持しつつテキスト化します。ただしPDFはWordと異なり文書構造が明示されていないため、**箇条書きや表形式のテキストは崩れる**可能性があります。抽出後にユーザーがプレビューで修正できる前提で、まずはプレーンテキストを得る実装とします。

構造化: 抽出したテキストから懸案データの構造（タイトル・詳細説明・対策内容）を判別します。ここで**「テンプレート前提」方式**と**「自由形式」方式**の2案があります。

- **テンプレート前提方式:** アップロードされるファイルがあらかじめ懸案データ用のフォーマットに沿っていることを仮定します。例えばExcelなら「A列:タイトル、B列:詳細、C列:対策」のように列が固定されたテンプレートを用意してもらい、システムは固定位置から値を取り出すだけなので構造化が容易です。同様にWordでも、「見出し1がタイトル、以下の段落が詳細、見出し2で'対策:'と書かれている箇所から対策内容」といった明確な書式ルールを決めておけば、その規則に従ってテキストを分割できます。この方式のメリットは**実装がシンプルで誤解析が少ない**点です[9]。デメリットは**利用者にフォーマット遵守を求める**ため運用上の制約が増えることと、フォーマットが異なる既存資料を取り込む際に調整が必要なことです。

- **自由形式方式:** ファイル内のテキストからプログラムで**見出しや区切りを検出**し、自動的にタイトルや対策部分を推定する方式です。例えば、「○○障害について」「対策：〜〜」といったキーワードや改行パターンから区切る、あるいは文書内の最初の1行をタイトル、それ以降を詳細と対策とみなす、など柔軟に対応します。メリットは**ユーザーが特定フォーマットに合わせる必要がない**点ですが、デメリットとして**解析ロジックが複雑**になり誤判定のリスクがあります。また高度なNLP処理（自然言語処理）が必要になる可能性もあります。初心者による実装難度を考えると、まずはテンプレート方式で確実に取り込めるようにし、将来的に必要に応じ自由形式もサポートする方針とします（設計上の判断）。

プレビュー・修正画面のUX: ファイル解析後、ユーザーには抽出結果のプレビューを表示します。例えばExcelで10件抽出できたら、10件分のタイトル・詳細・対策をテーブル形式で一覧表示し、それぞれ編集可能な入力欄に入れます。Word/PDFからの場合は、一つのファイルから1件または複数件の懸案が見つかったかでUIを変えます。1件のみ抽出ならその内容をフォームに当てはめ、複数懸案が含まれると推定した場合は区切って複数フォームを縦に並べます。ユーザーはそれらを確認し、足りない項目があれば入力し、不備があれば修正します。特にPDFでは箇条書きの「・」が抜けたり改行位置がおかしくなる可能性があるため、プレビュー時に目視確認してもらうことが重要です。プレビューUIには**元ファイルのページビュー**（PDFの場合）を横に表示する案もありますが、ブラウザ上でPDFレンダリングする実装は複雑なので、本システムではテキストのみ表示とします。元ファイルと差異が大きい場合は、一旦キャンセルして手動登録してもらう運用も想定します（割り切り）。

確定登録: プレビューOKでユーザーが「確定」ボタンを押すと、バックエンドにJSON形式で懸案データ一覧が送信され、DBに一括登録します。登録処理ではトランザクションを用い、全件正常に保存できたらコミット、途中でエラーがあればロールバックして「エラーが発生しました。○件目のデータをご確認ください」等のメッセージを返します。登録後、画面には「X件の懸案を登録しました」と結果を表示し、一覧画面へ戻す、あるいは新規登録した懸案のみをフィルタ表示するなどしてユーザーが後で確認しやすいようにします。

PDFがスキャン画像の場合の扱い: PDFにはテキストデータを含むものだけでなく、紙をスキャンした画像PDFも存在します。この場合、通常のテキスト抽出では何も得られません。対応策としてはOCR（Optical Character Recognition：光学文字認識）を組み込むことが考えられます。OCRエンジン（例えばTesseract）をサーバに導入し、ページごとに画像として処理して文字起こしすることになります。ただ、OCR結果は誤認識の修正が必要だったり、日本語の場合精度が英語より劣る課題があります。また初心者がOCR連携まで実装するのはハードルが高いため、本システムでは**スキャンPDFは非対応**とし、運用で回避することを前提とします（例えばそういった資料は一度手でテキスト化してから登録）。将来的に需要があればOCR組み込みを検討し、その際はOCR実行→テキストプレビュー→ユーザー修正というワークフローを追加します。尚、取り込み設定で「画像PDFの場合はエラーにする/警告を出す」といった挙動を定め、ユーザーに分かりやすいメッセージを出すようにします。

3.4 画像仕様（重要）

画像保存方式: 懸案に添付された画像は**ファイルシステム上に保存**します。データベースにバイナリ（BLOB）で直接保存する方法もありますが、ファイルサイズが大きくなるとDBの肥大化・バックアップ難度が上がるため避けます。代わりに、サーバ内の決められたディレクトリ（例: `attachments/`）にアップロード画像を置き、その**パスをDBに保持**します。こうすることで、アプリケーションからはパス経由で画像ファイルを読み書きでき、ウェブサーバから静的ファイルとして提供することも容易です。画像ファイル名はアップロード時に**ユニークな名前**に変更します（例えば「issue_<懸案ID>_<連番>.<拡張子>」等）。オリジナルのファイル名もDBに保存しておき、必要に応じユーザーに表示（ダウンロード時にオリジナル名を復元など）します。

画像メタ情報: 画像添付に伴い、以下のようなメタデータを保存します。 - **添付ID:** 各添付画像の一意なID。 - **ひもづく懸案ID:** どの懸案に属する画像かを示す外部キー。 - **ファイルパス:** サーバ内の保存先パス。またはベースディレクトリからの相対パス。 - **元ファイル名:** ユーザーがアップした元のファイル名（日本語名やスペース等は保存しておく）。 - **ファイル種別(MIME):** 画像の種類（image/png, image/jpegなど）。表示時のContent-Type設定やアイコン表示に使用。 - **ファイルサイズ:** サイズ上限をチェックする用途や、あとで容量集計する際に使用。 - **アップロード日時:** いつ添付されたか。運用上の追跡やソート表示に利用。 - **アップロード者:** どのユーザーが添付したか（懸案の登録者と同一なら省略も可）。 - **キャプション（説明文）:** ユーザーが入力した画像の説明テキスト。これは懸案詳細画面で画像の下に表示したり、検索対象データとして活用します。 - **OCRテキスト:** （任意）画像内の文字をOCRで抽出したテキスト。例えばエラーメッセージのスクリーンショットなら、その文字列をOCRし保存します。OCRはオプション機能であり、MVPでは実装せず必要に応じ追加します。

LLM連携時の扱い: 添付画像に関連する情報をLLMの回答にどう活かすかは課題です。現状使用するLLMはテキスト専用モデルであり画像そのものを理解できないため、画像から得られるテキスト情報を活用するか、存在を伝える程度に留まります。

- **OCRでテキスト化してプロンプトに入れる案:** 画像内に重要な文字情報がある場合（例: エラーメッセージのコードやログ画面）、OCRで抽出したテキストを懸案のテキストデータに含め、検索やプロンプト提供に利用します。例えば「（添付画像のOCR抽出結果：XXXXX）」のように懸案詳細に追記する運用です [8]。こうすれば検索にも引っかかりやすくなり、LLMが回答時にその文字列を参照できる利点があります。ただしOCR誤認識があると誤情報を与える懸念があるため、抽出結果はユーザーが確認・編集できるUI（OCR結果表示→編集→保存）を設けることが望ましいです。MVPでは一旦OCRはスキップし、**キャプションの活用**で代替します。ユーザーが画像アップ時に手入力したキャプション（例えば「エラーメッセージが表示されたダイアログ」等）を検索対象に含め、回答時にも参照データとしてLLMに渡すようにします。

- **メタ情報のみで画像添付を通知する案:** 画像自体の内容は使えないが、「この懸案には画像がある」という事実だけLLMに教えることも考えられます。例えばプロンプト内に「※この懸案には関連画像があります」と記述し、モデルに画像の存在を伝えます。モデルは詳細を知らなくても「詳細は添付

画像参照」と回答するかもしれません。しかし実際にはモデルはその画像を見られないため、有益な回答にはつながりにくいです。むしろユーザーが直接画像を見る方が早いので、AI回答では無理に触れず、UI側で画像の存在を表示（例えば回答メッセージの横にクリップアイコンを出しクリックで画像表示）するほうが現実的です。

- **将来のマルチモーダル対応:** 今後、画像も入力として解析できるマルチモーダル対応のLLM（画像理解AIとの連携など）が実用化された場合には、本システムも拡張を検討します。その際は画像ファイルをLLMに渡し、画像内容を説明させるようなプロンプトや、画像キャプション生成モデルとの組み合わせなどが考えられます。ただし2025年時点でローカルで軽量に動作する画像対応LLMは限られるため、まずはテキストベースの運用で十分か検証します。

以上を踏まえ、現段階では**画像添付はあくまで補助情報**として扱い、懸案データベースの主たる検索・回答にはテキスト情報を用いる方針です。画像自体はユーザーが必要に応じてUIから直接確認する運用とします。

4. 技術選定（初心者向け）

本システムを実装する技術スタックとして、初心者の習熟しやすさやWindows環境への導入容易性を考慮し、以下の3案を検討しました。

- **案A:** Node.js（サーバ） + Express（Webフレームワーク） + SQLite（DB） + React（フロントSPA） + Ollama（LLMサーバ）
- **案B:** Python（サーバ） + FastAPI（Webフレームワーク） + SQLite（DB） + React（フロントSPA） + Ollama（LLMサーバ）
- **案C:** .NET（C#サーバ） + ASP.NET Core（Webフレームワーク） + SQLite（DB） + React（フロントSPA） + Ollama（LLMサーバ）

それぞれについて、以下の評価軸で比較しました。

評価軸	案A: Node.js/Express	案B: Python/FastAPI	案C: .NET Core (C#)
学習コスト	JavaScriptに慣れていれば低め。Expressはシンプルだが周辺知識（npm管理など）が必要。フロントと言語統一できる利点。	Pythonは文法が平易で初心者に人気。FastAPIは宣言的にAPIを定義でき理解しやすい。AI関連ライブラリ豊富でサンプル多数。	C#は文法が厳密で初心者にはややハードル高。Visual Studio等IDE前提だが自動生成支援である程度カバー可。
Windows導入容易性	Node.jsは公式インストーラで導入簡単。特別な依存も少なくWindowsでそのまま動作。	PythonもWindows対応良好だが、環境変数や仮想環境管理に慣れが必要。Ollama自体はWindowsネイティブ非対応のためWSL等が別途必要。	.NETはWindows標準のように統合されており導入容易。ただしOllamaとの連携に工夫要（外部プロセス呼出など）。
同時100ユーザー性能	Node.jsは非同期I/Oが得意で高並行処理に強い。ただしLLM処理は別スレッド化が必要（Node本体はシングルスレッド）。ワーカープロセスの管理など追加実装が必要となる。	FastAPIは非同期処理対応で高負荷でも比較的安定。マルチスレッド/マルチプロセスをGunicorn+Uvicornでワーカーを増やしてさばける。Python自体は遅めだがLLM処理は外部なので問題なし。	.NET Coreは高性能でスレッド管理も自動、高並行にも強い。非同期も容易に書ける。ただしLLM処理部分は結局単体逐次なので全体性能は他案と大差ない見込み。

評価軸	案A: Node.js/Express	案B: Python/FastAPI	案C: .NET Core (C#)
LLM連携の実装容易性	Node.jsからOllamaを使うにはHTTPリクエストを送るか、CLIコマンド実行で結果を受け取る形。公式SDKは無いがfetchでREST呼び出し可能。AI関連はPythonに比べサンプル少。	PythonはLLM・RAG周りのライブラリが豊富[10]。OllamaもHTTP API経由で簡単に使えるし、LangChain等統合ライブラリも利用可能。RAG実装例もPythonが多く初心者参考情報が多い。	.NETからOllama REST APIを叩くことは可能（HttpClient利用）。ただAI分野のOSSライブラリはPython主体で、C#だと自前実装が増える。Gabriel氏のOllama+Qdrant RAG記事など一部情報あり[11]。
コミュニティ情報量	Web開発全般ではNode/Expressの情報豊富。ただし「ローカルLLM×Node」の組み合わせ事例はやや少ない。フロントと統一言語で書ける点は初学者にメリット。	Python/FastAPIは近年人気でドキュメント・記事多数。特にAI/データ分析系はPython一択の面があり、日本語情報も多い。初心者が困った時検索で解決しやすい。	.NET/C#はMicrosoft公式の充実ドキュメントや大企業での実績は豊富。ただAI組込み例は限定的。日本語コミュニティもあるが、OSS文化はNode/Pythonに比べやや弱い印象。
保守性	JavaScriptは動的型ゆえに大型化すると保守が難しくなる場合も。TypeScriptを使えば改善するが初心者には負荷増。パッケージも頻繁に更新されるため追随が必要。	Pythonはシンプルで読みやすく、少人数開発には適する。型チェックは弱いがコード量が比較的少なく書けるため保守もしやすい。FastAPIはコード構成も整理されテストもしやすい。	C#は静的型付けでIDEの支援もあり大規模開発向き。保守性は高いが、その分初期実装量や概念（DIコンテナ等）が多く初心者には難解な部分も。

この比較から、**案B: Python + FastAPI** を最終採用技術とします。理由は、**AI・データ処理の実装容易性と情報の豊富さ**です。Pythonは自然言語処理やLLMの扱いに強く、Ollamaとの連携やRAG実装についてもサンプルコードやライブラリが充実しています[10]。初心者がつまづきにくく、Windows環境でも大きな問題なく動作させられる点も考慮しました（Ollama自体は現状Windows未対応ですが、WSL2上で動かす手順が確立されています[12]）。Node.js案も魅力的でしたが、主にLLMまわりの実装ノウハウが少ない点で見送りました。C#案は開発効率よりも学習コストが懸念されるため今回の用途では採用を見送ります。

フロントエンド: なおフロントエンドは3案ともReact（もしくはその周辺のモダンJSフレームワーク）を想定しています。初心者にReactは少し高度ですが、既成のUIコンポーネントライブラリを使えばフォームや一覧画面を比較的簡単に構築できます。加えて、将来要件が増えてもReact + REST APIの構成は拡張性があります。UIの実装に時間を割きすぎないよう、必要ならMaterial-UIやBootstrap等を利用し効率化します。

ローカルLLMモデル候補: 本システムの中核となるLLMについては、**日本語対応**と**軽量でローカル動作可能**なモデルを選ぶ必要があります。以下の観点でモデルを2段階提案します。

- **推奨モデル（初期）:** 小〜中規模で、CPU単体や低スペックGPUでも動作するモデル。
- **上位モデル（高精度）:** ハードウェアに余裕がある場合に導入する大規模高性能モデル。

具体的には、初期導入として**Mistral 7B**や**Llama2 7B Chat**など7B（70億パラメータ）クラスのモデルを推奨します。Mistral-7Bは2023年に公開されたオープンモデルで、**Llama2 13Bを上回る性能**を示したことで注目されました[13]。パラメータ数が少ない分、モデルサイズも小さく（4bit量子化で約4~5GB）、16GB程度のRAMがあればCPUのみでも動かせる点が魅力です。日本語もある程度理解可能で、社内QA用途に耐えると考

えられます。Llama2　7Bも安定したモデルで、こちらはMeta社提供の英文ベースですが指示調整版は多言語対応力があります。

PCスペックが高く、より精度を求める場合は**Llama2　13B**や**Japanese-Llama系モデル**を検討します。例えばLlama2　13Bは7Bより大きなモデルであり文脈理解や回答の一貫性が向上します。GPU（例えば24GB VRAMクラス）を使用すれば実用的な応答速度が得られるでしょう。日本語特化の例としては、**ELYZA社の日本語LLM（8B）**があります。ELYZAの8Bモデルは日本語性能がGPT-3.5並みと評価されており[14]、社内文書のQAにも適すると期待されます。CyberAgentの**OpenCALM**や**CyberAgentLM　22B**、Lightblue社の**ao-Karasu 72B**といった国内モデルもありますが、22B以上は動作環境のハードルが高い（RAM数十GB級、GPU必須）ため、まずは扱いやすい7B〜13B級で構築し、後にモデル差し替えで精度向上を図る戦略を取ります。

また、モデルを選ぶ際にはシステムの**メモリとCPU/GPU能力**に合わせる必要があります。一般的なオフィスPC（例えばRAM16GB、CPUのみ）なら7Bモデル程度が現実的です。ゲーミングPCなど高性能GPU（例えばNVIDIA RTX 4090など24GB以上）を搭載できる場合、13B〜30Bクラスも視野に入ります。70Bモデルは現状複数GPUが必要になるため社内常設システムには難しいでしょう。**初期導入**では確実に動く軽量モデルで全体を検証し、その後必要に応じ**上位モデル**にスイッチできるよう、モデル差し替え手順を確立しておきます。

5. データモデル設計

システムで扱う主なエンティティ（データのまとまり）とその関係を定義します。以下に想定エンティティと概要を示します。

- **Users（ユーザー）:** システム利用者情報（ユーザーID、名前、パスワードハッシュ、権限ロール等）。※簡易運用なら単一管理者のみでテーブル不要の可能性もありますが拡張性のため定義。
- **Issues（懸案）:** 過去の懸案事項そのもの。懸案ID、タイトル、詳細説明（問題内容）、対策内容、登録日時、登録者ID、タグなどを含みます。
- **IssueAttachments（懸案添付）:** 懸案に紐づく添付ファイル（主に画像）。添付ID、懸案ID（外部キー）、ファイルパス、ファイル種別、キャプション、アップロード日時等。
- **ImportedFiles（インポートファイル）:** ファイル取り込み履歴。アップロードしたファイルごとに記録を保持（インポートID、ファイル名、アップロード日時、アップロード者、取り込んだ懸案件数など）。
- **ChatHistory（チャット履歴）:** ユーザーのQA履歴。チャットID、ユーザーID、質問テキスト、回答テキスト、日時、参照懸案（後述）など。
- **ChatRef（チャット参照）:** チャットの回答が参照した懸案を関連付ける中間テーブル。ChatHistoryとIssuesを多対多で紐づけ、フィールドはチャットID、懸案ID。

上記のリレーションをER図（エンティティ関係図）で示します。

```
erDiagram
    %% エンティティ定義
    USERS ||..o{ ISSUES : "登録"
    USERS ||..o{ IMPORTED_FILES : "アップロード"
    ISSUES ||--o{ ISSUE_ATTACHMENTS : "添付"
    IMPORTED_FILES ||--|{ ISSUES : "取り込み生成"
    USERS ||--o{ CHAT_HISTORY : "質問"
    ISSUES ||--o{ CHAT_REF : "参照された"
    CHAT_HISTORY ||--o{ CHAT_REF : "参照"
```

```
%% エンティティ詳細定義
USERS {
    int user_id PK
    string username
    string password_hash
    string role  // 'admin' or 'user'
}
ISSUES {
    int issue_id PK
    string title
    text description
    text resolution  // 対策内容
    string tags  // カンマ区切りタグ（任意）
    datetime created_at
    int created_by FK -> USERS.user_id
    int imported_from FK -> IMPORTED_FILES.import_id
}
ISSUE_ATTACHMENTS {
    int attachment_id PK
    int issue_id FK -> ISSUES.issue_id
    string file_path
    string file_name_original
    string file_type  // MIMEタイプ
    int file_size
    string caption
    datetime uploaded_at
    int uploaded_by FK -> USERS.user_id
}
IMPORTED_FILES {
    int import_id PK
    string file_name
    string file_type
    int file_size
    datetime imported_at
    int imported_by FK -> USERS.user_id
    int issue_count  // 何件のISSUEを生成したか
}
CHAT_HISTORY {
    int chat_id PK
    int user_id FK -> USERS.user_id
    text question
    text answer
    datetime asked_at
}
CHAT_REF {
    int chat_id FK -> CHAT_HISTORY.chat_id
    int issue_id FK -> ISSUES.issue_id
    %% 複合PK (chat_id, issue_id)にしてもよい
}
```

（※ER図中の記法: `||` は1（主キー側）、 `o{` は0個以上の関連（子テーブル側）を示します。）

エンティティ説明:

- **Users:** ユーザー情報。最小限としてユーザー名、パスワードハッシュ（認証用）、権限ロールを持ちます。ロールは `'admin'`（管理者）か `'user'`（一般ユーザー）で、一部機能の制御に用います。パスワードは平文では保持せずソルト付きハッシュ化します。ユーザーIDは主キー（自動採番）。

- **Issues:** 懸案データの本体。タイトル（件名）、問題の詳細説明、取った対策（解決方法）のテキストを主に持ちます。タグは簡易実装としてカンマ区切りの文字列で格納します（正規化するならIssueTagsのような別テーブルだが簡便さ優先）。`created_by` で登録者（Usersテーブル参照）を保持します。`imported_from` はもしファイルから取り込まれたデータならどのImportedFiles経由かを示す外部キーです（手入力ならNULL）。これにより、どのファイルから作成された懸案か後追いできます。

- **IssueAttachments:** 懸案に紐づく添付ファイル。基本的に画像を想定しますが、将来的に他のファイル形式も許容するならフィールド設計はそのままで対応可能です。`file_path` に保存先パス、`file_name_original` に元のファイル名、`file_type` にMIMEタイプ、`file_size` にバイトサイズを格納します。`caption` はユーザー入力の説明文です。`uploaded_by` でアップロード者（通常は懸案の登録者と同一かもしれませんが一応別に記録）。インデックスとしてissue_idに付与し、懸案ごとの添付一覧取得を高速化します。

- **ImportedFiles:** ファイル取り込みの履歴。こちらは主にログ用途で、アップロードしたファイルの名前・サイズ・種類、日時、実行者、そこから何件の懸案が生成されたか（issue_count）等を残します。実際に取り込んだ懸案IDとのリレーションはIssuesのimported_fromで辿れるため、ImportedFilesからIssuesへの1対多関係になります。将来的に同じファイルを2度取り込んだケース等にも備え、ImportedFilesを残しておくと重複登録防止にも使えます（ファイルハッシュを計算し、過去に同じハッシュのファイルをインポート済なら警告するなど発展可能ですが、現段階では考慮外）。

- **ChatHistory:** チャットの質問と回答の履歴です。ユーザーID、質問テキスト、回答テキスト、日時を保存します。1レコードでQとA両方持つ構造にしていますが、チャットの各ターンを厳密に管理するならuserメッセージとassistantメッセージを別テーブル/フィールドにする方法もあります。今回は1対1のQ&Aペアごとに記録する簡易形とし、会話の履歴はasked_atでソートすることで再現します（1質問につき1回答という前提）。会話の連続性をもたせる場合はsession概念を入れることもできますが、必須ではないので省略します。

- **ChatRef:** あるチャット質問の回答が参照した懸案のリストを保持する中間テーブルです。例えばChatHistory 5番の回答がIssue 2とIssue 8を参考にして生成されたなら、本テーブルに(5,2)と(5,8)の2行が入ります。これにより、チャット履歴から関連懸案を引けるし、逆に懸案から「ChatHistoryで使われた例」を集計できます。主キーは(chat_id, issue_id)の複合でも良いですが、単純にオートIDにしても機能上問題ありません。MVP実装では省略しても基本機能に支障はないため、開発スケジュール次第で後回しにします（設計上の柔軟性）。

テーブル定義例（SQLite）: 各テーブルのDDLをSQLite風に記述します。

```
-- Users table
CREATE TABLE users (
  user_id    INTEGER PRIMARY KEY AUTOINCREMENT,
  username   TEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL,
  role       TEXT CHECK(role IN ('admin','user')) DEFAULT 'user'
```

```
);

-- Issues table
CREATE TABLE issues (
  issue_id    INTEGER PRIMARY KEY AUTOINCREMENT,
  title       TEXT NOT NULL,
  description  TEXT,
  resolution   TEXT,
  tags        TEXT,
  created_at   DATETIME DEFAULT CURRENT_TIMESTAMP,
  created_by   INTEGER,
  imported_from INTEGER,
  FOREIGN KEY(created_by) REFERENCES users(user_id),
  FOREIGN KEY(imported_from) REFERENCES imported_files(import_id)
);

-- Index for full-text search on issues (using FTS5 virtual table)
CREATE VIRTUAL TABLE issues_fts USING fts5(title, description, resolution, content='issues',
content_rowid='issue_id');

-- Trigger to update FTS table on issues insert/update/delete (if needed)
-- (SQLite FTS5 can use triggers or the content option with external content table)

-- IssueAttachments table
CREATE TABLE issue_attachments (
  attachment_id   INTEGER PRIMARY KEY AUTOINCREMENT,
  issue_id      INTEGER NOT NULL,
  file_path     TEXT NOT NULL,
  file_name_orig  TEXT,
  file_type     TEXT,
  file_size     INTEGER,
  caption       TEXT,
  uploaded_at    DATETIME DEFAULT CURRENT_TIMESTAMP,
  uploaded_by    INTEGER,
  FOREIGN KEY(issue_id) REFERENCES issues(issue_id) ON DELETE CASCADE,
  FOREIGN KEY(uploaded_by) REFERENCES users(user_id)
);
CREATE INDEX idx_attach_issue ON issue_attachments(issue_id);

-- ImportedFiles table
CREATE TABLE imported_files (
  import_id    INTEGER PRIMARY KEY AUTOINCREMENT,
  file_name    TEXT,
  file_type    TEXT,
  file_size    INTEGER,
  imported_at  DATETIME DEFAULT CURRENT_TIMESTAMP,
  imported_by  INTEGER,
  issue_count  INTEGER,
  FOREIGN KEY(imported_by) REFERENCES users(user_id)
);
```

```
-- ChatHistory table
CREATE TABLE chat_history (
  chat_id   INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id   INTEGER,
  question  TEXT,
  answer    TEXT,
  asked_at  DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY(user_id) REFERENCES users(user_id)
);
CREATE INDEX idx_chat_user ON chat_history(user_id);

-- ChatRef table
CREATE TABLE chat_ref (
  chat_id  INTEGER,
  issue_id INTEGER,
  FOREIGN KEY(chat_id) REFERENCES chat_history(chat_id) ON DELETE CASCADE,
  FOREIGN KEY(issue_id) REFERENCES issues(issue_id) ON DELETE CASCADE
  -- PRIMARY KEY(chat_id, issue_id) -- (optional composite PK)
);
CREATE INDEX idx_ref_issue ON chat_ref(issue_id);
```

インデックス方針: - Issuesに関しては全文検索用にFTS5仮想テーブル `issues_fts` を使います [6] 。これによりタイトル・詳細・対策の各フィールドからキーワード検索が高速に行なえます。通常の索引も、`issue_id`（主キー）にはデフォルト、`created_by` や `imported_from` は外部参照頻度次第で付与検討します（例えば特定ユーザーの登録懸案一覧を出す場合は `created_by` 索引が有用）。 - IssueAttachmentsはissue_idでの検索（懸案ごとの添付取得）が多いため、`idx_attach_issue` を付与しています。 - ChatHistoryはuser_idで絞って履歴を見る場合があるため `idx_chat_user` を設定しました。チャット履歴全検索は基本しない前提なのでasked_atは索引不要です（降順取得は主キー降順で代替）。 - ChatRefはissue_idで「この懸案が参照されたチャット一覧」を引くクエリを想定し、`idx_ref_issue` を付けています。chat_idで引く（チャットから参照懸案を得る）のはChatHistory取得後にJOINする形で高速に取れるため、場合によっては `chat_id` にも索引追加します。 - その他外部キー制約にON DELETE CASCADEを付けた箇所があります（例えばIssue削除時に添付とChatRefも削除）。これはデータ整合性の担保ですが、運用上Issue削除を基本しないなら付けなくても構いません。今回は念のため整合性を重視する設計としています。

添付ファイルの扱い（保存パス等）： `file_path` には、サーバ内での相対パス（例: `"attachments/issue_12_img1.png"`）を保存します。こうすることでサーバ移転時でもベースディレクトリさえ合わせればパスが有効になります。また、別途**ハッシュ値**（MD5やSHA1など）をファイルアップロード時に計算し保存しておけば、同一ファイルの重複検出や改ざんチェックにも使えます。今回は必要性低と判断しテーブル設計には入れていませんが、将来的にImport時の重複確認などで活用できるでしょう。

インポート履歴: ImportedFilesとIssues間の関連はIssues側にimported_fromを持つため、どのファイルから取り込んだかを後から分析可能です。例えば「過去にimport_id=5のExcelから取り込んだ懸案を全部検索」といったこともできます。またImportedFiles.issue_countでファイル毎の登録件数を記録しているので、「Excel_A.xlsxは10件の懸案を登録した」等のログも見られます。インポート時は、ファイルのハッシュ（CRC等）を取って同じファイルの二重処理を防ぐことも検討しましたが、そこまでの要件は無いため省いています。運用上、ユーザーが同じファイルを再アップロードしないよう注意喚起する程度で十分でしょう。

6. API設計

フロントエンド（React SPA）とバックエンド（FastAPI or Node API）の間はRESTfulなJSON APIで通信します。主要なエンドポイントを以下にまとめます。

エンドポイント (URI)	メソッド	説明	認証	主なリクエスト例 / レスポンス例
ユーザー認証系				
`/api/login`	POST	ログイン認証（必要なら実装）	（未定）	リクエスト: `{username, password}` レスポンス: 成功時200 + セッショントークン/JWT
`/api/logout`	POST	ログアウト	（未定）	（サーバ側セッション無効化、Cookieクリア等）
懸案 (Issue) 関連				
`/api/issues`	GET	懸案一覧取得	要ログイン (admin/user)	**レスポンス:** 懸案オブジェクト配列（最新20件など、ページング対応）
`/api/issues?query=...`	GET	懸案検索（キーワード）	要ログイン	**レスポンス:** クエリにマッチした懸案一覧（全文検索の上位N件）
`/api/issues/<id>`	GET	懸案詳細取得	要ログイン	**レスポンス:** 懸案オブジェクト詳細（添付一覧・関連情報含む）
`/api/issues`	POST	懸案新規登録	要ログイン	**リクエスト:** 懸案JSONデータ（title, description, resolution等）
`/api/issues/<id>`	PUT	懸案編集	要ログイン（権限チェック）	**リクエスト:** 更新フィールドJSON（タイトルや内容）
`/api/issues/<id>`	DELETE	懸案削除	要管理者	**レスポンス:** 成功時ステータス200（内容なし）
懸案ファイル取り込み				
`/api/issues/import`	POST	ファイル取り込みプレビュー	要ログイン（adminのみ想定）	**リクエスト:** フォームデータ（ファイル添付） **レスポンス:** 抽出結果プレビューJSON
`/api/issues/import/confirm`	POST	ファイル取り込み確定登録	要ログイン（admin）	**リクエスト:** 抽出後の懸案データリストJSON **レスポンス:** 登録結果（件数や生成ID）
添付ファイル (画像) 関連				
`/api/issues/<id>/attachments`	GET	添付ファイル一覧取得	要ログイン	**レスポンス:** 添付メタデータ配列（各画像のURLやキャプション等）

エンドポイント (URI)	メソッド	説明	認証	主なリクエスト例 / レスポンス例
`/api/issues/<id>/attachments`	POST	添付ファイルアップロード	要ログイン	**リクエスト:** フォームデータ（画像複数可） **レスポンス:** アップロード結果（添付ID・URLリスト）
`/api/attachments/<attId>`	DELETE	添付ファイル削除	要ログイン（権限）	**レスポンス:** 成功ステータス（ファイル削除実行）
チャット (QA) 関連				
`/api/chat/question`	POST	質問送信→AI回答取得	要ログイン	**リクエスト:** `{ "question": "〜?" }` **レスポンス:** `{ "answer": "回答テキスト...", "references": [関連懸案ID,...] }`
`/api/chat/history`	GET	チャット履歴一覧取得	要ログイン	**レスポンス:** ユーザー自身の過去Q&A一覧（日時降順）
`/api/chat/history/<id>`	GET	特定チャット履歴詳細	要ログイン	**レスポンス:** 指定IDのQ&A詳細＋参照懸案
管理者用				
`/api/admin/config`	GET	システム設定取得	管理者のみ	**レスポンス:** 設定値（モデル名、温度、トークン上限 等）
`/api/admin/config`	PUT	システム設定更新	管理者のみ	**リクエスト:** 設定値JSON（変更項目のみでも可）
`/api/admin/users`	GET	ユーザー一覧取得	管理者のみ	**レスポンス:** ユーザー情報リスト（パスハッシュ除く）
`/api/admin/users`	POST	ユーザー新規登録	管理者のみ	**リクエスト:** 新ユーザー情報JSON（username, role等）
`/api/admin/users/<id>`	PUT	ユーザー情報更新	管理者のみ	**リクエスト:** 更新内容JSON（role変更等）
`/api/admin/users/<id>`	DELETE	ユーザー削除	管理者のみ	**レスポンス:** 削除成功ステータス

（※URIは先頭に`/api`を付け、フロント側のルーティングと区別しています。）

上記は主要なAPIの一覧で、認証が必要なエンドポイントには「要ログイン」と記載しています。シンプルな運用ならログイン無しで全機能使える設定も可能ですが、**懸案データの改ざん防止や設定変更は管理者のみ**としたいため、基本的にはログインセッションを前提としています。

重点API詳細:

- **懸案CRUD:** たとえば新規登録API `/api/issues` (POST)は、リクエストボディに懸案情報(JSON)を受け取ります。例：

リクエスト:

```
{
 "title": "サーバ起動時にエラー発生",
 "description": "Windowsサーバでサービス起動時にXYZエラーコードが出る",
 "resolution": "設定ファイルのパスを修正し再起動することで解決した",
 "tags": "サーバ,起動,設定ミス"
}
```

レスポンス（成功時）:

```
{
 "issue_id": 123,
 "message": "Issue created successfully"
}
```

このAPIではバックエンドで認証済ユーザーIDを取得し、`created_by`にセットしてDB保存します。バリデーションとして必須項目titleの空チェックなどを行い、不備があれば400エラーを返します。

- **懸案検索:** `/api/issues?query=キーワード` (GET) はクエリパラメータ`query`で渡されたキーワードで全文検索を行います。バックエンドではSQLite FTS5のクエリを実行し、例えば`SELECT issue_id, title, snippet(issues_fts, ...) FROM issues_fts WHERE issues_fts MATCH 'キーワード' LIMIT 20;`のように検索します。レスポンスはシンプルにissue一覧JSONです。例:

```
[
 {
  "issue_id": 120,
  "title": "サーバ起動時にエラー発生",
  "snippet": "...サービス起動時に<b>エラー</b>コードが出る..."
 },
 {
  "issue_id": 85,
  "title": "エラーログ分析ツール",
  "snippet": "...<b>エラー</b>ログを収集して解析する方法..."
 }
]
```

（snippetはFTSのハイライト結果をHTMLで含む例。必要に応じシンプルな形に整形。）

ユーザーはこの結果一覧から目的の懸案詳細にアクセスできます。なおタグ検索を別途実装する場合は`/api/issues?tag=XYZ`のようなパラメータを設け、tagsカラムをLIKE検索するかTagテーブルからJOINする形になります。

- **ファイルアップロード→プレビュー→確定登録:** このフローは2段階APIで実現します。
- `/api/issues/import` (POST): リクエストはContent-Type: multipart/form-dataで、フィールド名例えば`file`でファイルバイナリを含めます。バックエンドではファイルを一時保存またはメモリ上で

処理し、上記3.3の**テキスト抽出**と**構造化**を行います。そして抽出結果を以下のようなJSONで返します。

レスポンス例:

```
{
 "import_id": 17,
 "file_name": "past_issues.xlsx",
 "parsed_issues": [
  {
   "temp_id": 1,
   "title": "サーバ起動エラー",
   "description": "Windowsサーバ...（詳細文）",
   "resolution": "設定を修正...（対策文）",
   "tags": ""
  },
  {
   "temp_id": 2,
   "title": "ログイン不具合",
   "description": "ユーザーがパスワードを...（詳細）",
   "resolution": "DBのユーザーテーブル...（対策）",
   "tags": "認証"
  }
 ]
}
```

`parsed_issues`は検出された懸案候補の配列です。各`temp_id`はクライアント側で編集識別用の仮IDです。クライアント（React）はこのJSONを受け取ったら画面にタイトル・詳細・対策を入力フィールドとして表示します。ユーザーが編集・確認し、問題なければ次を呼びます。

- `/api/issues/import/confirm` (POST): リクエストボディにユーザーが確定した懸案データー覧をJSONで送信します。このとき、`import_id`（先のレスポンスで発行したもの）も含めます。例えば:

```
{
 "import_id": 17,
 "issues": [
  {
   "title": "サーバ起動エラー",
   "description": "Windowsサーバでサービス起動時に障害が発生。詳細ログ: ...",
   "resolution": "設定ファイルのパス誤りを修正し再起動したところ正常起動した。",
   "tags": ""
  },
  {
   "title": "ログイン不具合",
   "description": "一部ユーザーが正しいパスワードでもログインできない事象が発生。",
   "resolution": "DBのユーザーテーブルに不整合があり修正。該当ユーザーは再ログイン成功。",
   "tags": "認証"
  }
```

```
 ]
}
```

バックエンドはこれを受け取りトランザクションを開始、issuesテーブルに2件INSERTします（それぞれimported_fromに17をセット）。問題なく全件追加できたらコミットし、ImportedFilesテーブルのimport_id=17レコードのissue_countを2に更新します。レスポンスは例えば:

```
{
 "message": "Imported 2 issues successfully",
 "issue_ids": [124, 125]
}
```

ここで取得した新規issue_idsを使って、フロントで「登録済み一覧に遷移」などの動きをします。なお、バリデーションエラー（必須タイトルが空など）があれば、どの項目にエラーかを示す情報を返し、フロントでユーザーに再修正を促すようにします。

- **画像添付アップロード:** `/api/issues/<id>/attachments` (POST)は、特定の懸案IDに画像を追加するエンドポイントです。リクエストはmultipart/form-dataで複数ファイル送信可能とします（`<input type="file" multiple>`から）。バックエンドでは懸案IDの存在確認をし、各ファイルを保存ディレクトリに配置、issue_attachmentsにレコードをINSERTします。レスポンスとしては、新規attachmentのIDとアクセスURL、ファイル名などを返します。例:

```
{
 "attachments": [
  {
   "attachment_id": 56,
   "url": "/uploads/issue_124_img1.png",
   "file_name": "error_screenshot.png",
   "file_type": "image/png",
   "size": 152345
  },
  {
   "attachment_id": 57,
   "url": "/uploads/issue_124_img2.png",
   "file_name": "settings.jpg",
   "file_type": "image/jpeg",
   "size": 53421
  }
 ]
}
```

フロントはこれを受け取り、画面上に画像プレビュー（`<img src="...">`）を表示します。URLは例えばExpressなら静的ファイル提供ミドルウェアで`/uploads/`パス以下をローカルディレクトリ`attachments/`にマッピングし、FastAPIなら`StaticFiles`で同様に設定します。
画像削除は認証と権限チェックを行い、issue_attachmentsからDELETEし、対応する実ファイルも削除します。削除API成功後はフロントで一覧からその画像を取り除きます。

- **チャット（RAG処理）:** `/api/chat/question` (POST)は、ユーザーからの質問を受けAIの回答を返すエンドポイントです。最も重要な処理であり、以下の手順を踏みます。

- リクエストJSONから`question`文字列を取得。例:
`{ "question": "以前発生したサーバ起動エラーの対策は何でしたか？" }`
- データベース（issues_fts）で質問文に含まれるキーワードを検索。【設計上簡略化】まずはキーワードマッチとし、例えば「サーバ」「起動」「エラー」などでマッチするIssueを全文検索します[6]。将来的には後述のベクトル検索で意味的に関連する候補を探すよう拡張します。
- 上位N件（例えば3件）のIssueデータを取得します。取得フィールドはタイトル・詳細・対策（場合によっては添付キャプションやOCRテキストも）。
- それらを元に**プロンプトを構築**します。プロンプト設計の例としては、Systemメッセージに「あなたは社内ナレッジベースを参照して質問に答えるアシスタントです。与えられた情報のみを使って回答してください。」などを与え、Userメッセージにユーザーの質問と**関連懸案情報**を含めます。関連情報のフォーマットは工夫が必要で、例えば：

```
ユーザー質問: <ユーザーの質問文>

以下は社内ナレッジから見つかった関連情報です：
[資料1] タイトル: サーバ起動エラー
問題: Windowsサーバでサービス起動時にエラーコードXXXが発生...
対策: 設定ファイルのパスを正しく設定し直した。

[資料2] タイトル: （別の懸案タイトル）...
（詳細・対策 要約 or 抜粋）
```

のように複数の資料を列挙し、それを踏まえて答えさせる形が考えられます。資料をそのまま長文で入れるとトークンを圧迫するため、**要約**や**抜粋**を入れる実装も検討します。初心者実装ではまずは丸ごと入れて様子を見るのもあり、長すぎる場合は先頭部分だけにするなど簡易対策でも良いでしょう。
- 構築したプロンプトをLLM(Ollama経由)に送り、テキスト生成を行います。OllamaのAPIではモデル名とプロンプトをPOSTすると応答が得られます。モデルがストリーム出力するならそれを受け取ってまとめます。
- 得られた回答テキストをそのままクライアントに返す前に、**解答内容の後処理**を行います。例えば、不適切表現がないか軽くチェックしたり（社内利用なのであまり神経質にならなくてよいが）、参照した資料を明示するため回答末尾に「(参照: サーバ起動エラーの対策)」のような出典を付与する案があります。モデルに出典を自力で書かせるのは難しいため、こちら側でChatRefに記録したIssueを基に文章を付け足すか、UIで別途リンク表示する方が簡明です。
- ChatHistoryテーブルにこのQ&AペアをINSERTします（`user_id`, `question`, `answer`, `asked_at`）。続いてChatRefにも参照IssueをINSERT（例: (chat_id=77, issue_id=120), (77, 85) 等）。
- 最終レスポンスJSONを組み立てます。MVP段階では最低でも`answer`テキストを返し、可能なら`references`として参照したissue_idリストやタイトルを含めます。例:

```
{
 "answer": "サービス起動時にエラーが発生した場合、設定ファイルのパスを修正することで解決しました。",
 "references": [
  { "issue_id": 120, "title": "サーバ起動エラー" }
 ]
}
```

ここでreferencesは回答根拠提示用ですが、UIではこれを見て「参照: サーバ起動エラー」と表示し、クリックでIssue #120詳細を開くといった機能を実装可能です。

フロントでは、POST後すぐに回答メッセージ枠を用意して「...thinking...」と表示し、レスポンスが届いたらそこにテキストを表示します。複数行の回答にも対応するため、メッセージ枠は可変高さでスクロール可能にします。

- **エラー設計:** APIはエラー時にHTTPステータスコードとエラー内容を返します。例えば、バリデーションエラーは400 Bad Requestで、レスポンスボディに`{ "error": "Title is required" }`のようなJSONを返します。認証が必要なAPIに未ログインでアクセスした場合は401 Unauthorizedを返し、フロント側でログイン画面にリダイレクトさせます。サーバ内部エラー（例: LLMプロセスが起動していない等）は500 Internal Server Errorで、詳細メッセージはログにのみ記録し、ユーザーには`{ "error": "Internal server error. Please contact admin." }`のように汎用メッセージを返します。
  一貫性のためエラーJSONフォーマットは統一し、例えば

  ```
  { "error": { "code": 400, "message": "〜エラー詳細〜" } }
  ```

  のようなネスト構造にしても良いでしょう。これによりフロント側でパースしやすくなります。

- **認証方法:** ローカルツールとはいえ複数ユーザー運用する際は簡易認証を入れます。方法は**Cookieセッション**か**JWT**の二択が考えられます。初心者に実装しやすいのは、おそらく**サーバサイドセッション**でしょう。具体的には、`/api/login`でユーザー名・パスワード検証後、サーバ側でセッションIDを発行しCookieに付与、以降のAPIでそのセッションIDをもとにユーザーを認識する方式です。これは状態をサーバ（メモリやSQLiteテーブル）で保持する必要がありますが、同一マシン内で閉じている本システムでは問題になりません。
  JWT（JSON Web Token）は、ステートレスにクライアント側（ブラウザLocalStorageなど）にトークンを保持します。メリットはAPIサーバが複数でも認証を共有できる点ですが、本システムは単一サーバなのでその利点は薄いです。一方で実装としてはサーバでの検証（署名確認）処理が必要ですが、ライブラリが整っているので難しくはありません。
  今回は**セキュリティ要件を緩和**し、まず**ログイン無し**でスタートすることも検討します（PC自体Windowsログインで管理されている前提で、ツール単体ではパスワードレスでもよいのではという判断）。しかし管理者機能まで含むため、最低限**管理画面に入るためのパスワード**設定はした方が安全です。そこで、**管理用パスワードを1つ決めて.env等に置き、管理画面アクセス時に要求する**簡易方式も案としてあります。これは実装は楽ですがユーザー個別認証ではないため、一律の鍵を知っているかで判定する形になります。
  最終的には、**ユーザー管理を導入する前提でCookieセッション方式を採用**し、管理者フラグのあるユーザーのみ管理APIにアクセスできるようミドルウェアでチェックする構成を想定します。トークンの期限やCSRF対策（SameSite属性など）は、ローカル用途なので大きな脅威ではありませんが、開発の余裕があれば実装・設定します。

7. RAG設計（重要）

本システムの回答生成は**Retrieval-Augmented Generation (RAG)** のアプローチを取ります。つまりLLM単体に全知識を詰め込むのではなく、**事前に関連データを検索で取得し、その情報をプロンプトに与えて回答させる**方法です。この章ではRAG部分の詳しい設計を示します。

参照対象データ: AIが回答時に参照できるデータとして、以下を組み込みます。 - **懸案の本文テキスト:** Issueレコードのタイトル、詳細説明、対策内容。これが主要なナレッジ源です。 - **添付画像のテキスト情報:** 前述のように、画像から抽出したOCRテキストや付与されたキャプションがあれば、それも懸案データの一部として検索・参照に用います。 - **過去チャット履歴:** 通常のRAGでは過去のQ&A履歴は知識源ではありません。ただ、ユーザーが続けて対話する場合は会話履歴を文脈として与える必要があります。今回、チャットは基本

1問1答形式を想定していますが、「先ほどの問題についてもう少し詳しく教えて」など**追問**がある場合、前の質問・回答内容を保持してプロンプトに含める必要が出ます。これも広義にはRAGと言えます。実装としては、ChatHistoryから直近の対話を一定数取り出し、システムメッセージに「前の会話: Q:... A:...」のように渡す方法が考えられます。ただし長い履歴はすぐトークン上限に達するため、**基本は各質問を独立**に扱い、必要に応じて直前Q&Aのみ文脈に入れる程度に留めます。

検索方式案: 1. **全文検索（キーワードマッチ）:** 初期実装ではSQLiteのFTS5を使ったキーワード検索を採用します [6]。ユーザー質問を形態素解析して単語単位に分け（簡易にはスペース区切り想定でもよい）、issues_ftsテーブルに対してMATCHクエリを実行します。例えば質問「サーバ エラー 対策」であれば、それらの語を含む文書をOR検索またはAND検索できます。スコア順で上位N件（デフォルト3件程度）を選びます。全文検索はシンプルで軽量ですが、**表記揺れや言い換えに弱い**欠点があります。例えば「ログインできない」と「サインイン不可」は文字的には別なのでヒットしません。しかし当面は運用でユーザーがキーワードを工夫する想定で割り切ります。

1. **ベクトル検索（Semantic検索）:** 将来的改善として、文書を**ベクトル埋め込み**し類似度検索を行う方法を検討します。具体的には、各懸案の内容をembeddingモデルでベクトル化し、ベクトル格納・検索ライブラリ（FAISSやChromaなど軽量なベクトルDB）に保存します。ユーザー質問も同じモデルでベクトル化し、近い距離の懸案ベクトルを検索することで、言葉の違いを超えたマッチングが可能になります [10]。例えば「ログイン不可」で検索して「サインイン障害」の懸案がヒットするなどの効果が期待できます。実装としてはFAISSをPython内に組み込むか、Chromaなどを使う案があります。1000件程度であればベクトルサイズ次第ですがメモリ数十MB程度に収まり、十分ローカルPCで扱えます。OllamaはembeddingAPIも持つようなので、それを活用する手もあります [4]。ただ初心者がいきなりベクトル検索を実装するのはハードルが高いため、**まずは全文検索で実現し、次ステップとしてベクトル検索モードを追加**する構想とします。拡張時は2通りの検索を切替可能にすると良いでしょう（設定で“検索方法: keyword or vector”を選択できる等）。

プロンプト組み立て方針:　モデルに渡すプロンプトには、**System**ロールと**User**ロール（必要ならAssistantロールの例も）を適切に使い分けます。システムメッセージでAIの振る舞いを制御し、ユーザーメッセージに質問と知識情報を含めます。例を示します。

```
<System role>
あなたは社内問い合わせ対応AIです。支給する社内ナレッジデータに基づいて、ユーザーの質問に日本語で簡潔かつ正確に答えてください。ナレッジに無いことは無理に作り出さず、「分かりません」と答えて構いません。

<User role>
質問: サーバ起動時にエラーが発生した場合の対策を教えてください。

社内ナレッジ:
[懸案1]
タイトル: サーバ起動エラー
詳細: Windowsサーバでサービス起動時にエラーコード0x1234が発生し、サービスが起動できない問題。
対策: 設定ファイルのパスに誤りがあったため、正しいパスに修正しサービスを再起動したところ正常に起動した。

[懸案2]
タイトル: 別サービスの起動不良
詳細: （以下略）
対策: （以下略）
```

回答を出力してください:

このように、ユーザー質問の後に"社内ナレッジ"として関連懸案の内容を提示しています。モデルはこれらを参考に回答することになります。ポイントは、**明示的にこの情報のみを使うよう指示**していることです。Systemプロンプトで「与えた情報のみを使え」と伝えることで、モデルの暴走（関係ない事柄の想像）を抑制します。ただしモデルによってはまだ知識補完しようとするかもしれず、完全防止は難しいですが、比較的制御は効きます。

出典（引用）提示: モデルの回答に出典を含めさせたい場合、「回答の末尾に参照した懸案番号を括弧書きしてください」という指示をSystemに入れることも考えられます。しかし多くの既存モデルはそのような形式指定に弱く、正確に番号を出す保証はありません。そこで**システム側でAfter-processing**する案を採ります。ChatRefに格納したissueを基に、回答テキストの末尾に例えば「(参考:　サーバ起動エラーの懸案)」と付与する、もしくは別UI要素で「参考資料: サーバ起動エラー 等」と表示する方針です。これならモデルの挙動に依存せず出典表示できます。

トークン制限への対応: LLMには入力長・出力長の上限があります。例えば7Bモデルでは4096トークン前後が一般的です。長文の懸案データを3件も入れると簡単に数千トークンになり、質問文やシステムメッセージ分と合わせてオーバーしかねません。この対策として: - **要約/圧縮:** 懸案データが長い場合、重要部分だけ抽出してからプロンプト挿入します。例えば「...（詳細は省略）」のように一部カットしたり、あらかじめ対策のポイントだけ書くことも可能です。ただし自動要約はモデルにやらせると本末転倒なので、例えば対策内容だけに限定する、などルールを決めて情報圧縮します。 - **上位N件に限定:** 初期は3件程度に絞るようにしますが、場合によっては関連度スコアが近い上位2件のみにしたり、逆に情報が足りなければ5件に増やすなど調整します。質問文の傾向である程度Nを変えるロジックも考えられますが複雑なので、ひとまず固定件数＋場合により手動設定で十分でしょう。 - **チャンク化:** 1件の懸案の内容が非常に長い場合（例えば長文マニュアルそのまま一件に入っているなど）、その中で質問に関連する部分だけ抜粋する工夫が必要です。これにはembeddingでの部分検索や、あらかじめ段落単位で区切っておき検索する手法があります。しかし1000件規模かつ1件平均そこまで長大ではない想定なので、現段階では対応不要と考えます。どうしても長い場合、全体を載せず「詳しくは懸案XX参照」など回答に誘導させるでもよいでしょう。

RAG処理フローまとめ: 前述を順に整理すると、RAGのシーケンスは以下です（シーケンス図(4)参照）。まずバックエンドで検索→上位データ選出→プロンプト構築→LLM応答取得→回答返却という流れです。この一連の処理をモジュール化し、例えば `getAnswerForQuestion(questionText)` のようなサービス関数にまとめます。内部では検索部分をStrategyパターンで差し替え可能（keyword vs vector）にしておくと拡張性が高いです。LangChainなどのフレームワークを使えば似たパイプラインを簡潔に書けますが、初心者が内部を理解するため自前で実装してみるのも良い経験でしょう。LangChain導入は将来、複雑なプロンプトやツール呼び出しを行う段階になってから検討します。

最後に、**LLM応答の品質管理**について触れます。RAGとはいえ、モデルが与えた情報を誤用したり、曖昧な質問に変な答えをするリスクは残ります。運用上は回答内容をユーザー自身が検証できるよう、参照した懸案のリンクを提示し**根拠を確認できるUX**を提供することが重要です。また、あまりにも見当違いの回答が来る場合は、プロンプトの改善（指示を強める）やモデル変更を検討します。これらは非機能側のチューニング要素となります。

8. UI/UX設計

ユーザーインターフェースはReactによるシングルページアプリ(SPA)で構築します。画面遷移を伴わず、React Router等で仮想的にページを切り替える方針です。主な画面とUI要素を整理します。

画面一覧: 1. **ログイン画面**（必要な場合のみ）
ユーザー名・パスワード入力フィールド、ログインボタン。シンプルなレイアウトで、認証成功後メイン画面へ遷移。（ログイン省略時はこの画面はなし） 2. **メイン画面（ダッシュボード）**
アプリの起点。ヘッダーナビゲーション＋メインコンテンツ領域で構成。ヘッダーにメニュー（「懸案一覧」「チャット」「設定」など）を配置。初期表示としては懸案一覧またはチャット画面を表示。 3. **懸案一覧画面**
過去の懸案をリスト形式で表示。検索バー（テキスト入力＋検索ボタン）を上部に配置。新規懸案登録ボタン、インポートボタンも配置。リストは表形式（タイトル、登録日、登録者、タグ等の列）で表示。各行をクリックすると詳細画面へ。行末に編集・削除アイコンも設置（権限ある場合のみアクティブ）。 4. **懸案詳細画面**
懸案の詳細内容を表示。タイトルは大見出し、以下に詳細説明と対策内容を段落区切りで表示。関連タグがあればタグ一覧表示（バッジ等）。添付画像がある場合はサムネイル一覧を下部に表示し、クリックで拡大ポップアップ。画面下部に「編集」「削除」ボタン（権限次第）。また「この懸案はチャット回答で○回参照されました」のような情報を付記することも検討（ChatRefから集計。ただし必須ではない）。 5. **懸案登録/編集画面**
フォーム形式。項目はタイトル（テキスト）、詳細説明（テキストエリア）、対策内容（テキストエリア）、タグ（テキスト、カンマ区切り入力支援）等。登録時には添付画像もここからアップロード可能なので、画像添付用の入力（ファイル選択ボタン＋選択済ファイル一覧のプレビュー）がある。編集時は既存内容をフォームに表示、添付一覧も表示し、削除ボタンも各画像につける。フォーム下に「保存」「キャンセル」ボタン配置。バリデーションは必須タイトルのチェックと、文字数超過チェック（例えばタイトル100文字以内など）をonChangeまたはonSubmit時に行い、エラー箇所は赤枠＋メッセージ表示。 6. **取り込みプレビュー画面**
ファイル選択後に表示される画面。アップロード直後はまず抽出中インジケータ（スピナー等）を出し、結果が来たらプレビュー表示。プレビューは複数懸案エントリを上下に並べたミニフォーム群として構成。例えばカードUIにして、それぞれタイトル・詳細・対策を小さなテキストエリアで表示/編集可能にする。各カードの横にチェックボックスがあり、「登録する」かどうか選択できるようにしてもよい（デフォルト全選択）。ユーザーは必要に応じテキストを修正。全部確認したら画面下の「登録実行」ボタンを押す。キャンセルボタンで中止（アップロードしたファイルのimport_idは削除するか無効扱い、DB残しても害はない）。操作性を高めるため、テキストエリアは一定高さでスクロール可能にし、長すぎる場合もUIが崩れないようにする。また各エントリに「元データ表示」リンクをつけ、クリックするとモーダルで該当部分の原文（例えばPDFのテキストブロックやWord段落）を表示することも考えられるが、実装コストのため見送り。**UX重視点:** 取り込みプレビューはユーザーにとって少し手間なので、エントリが多い場合にスクロール固定ヘッダーで見出しを残す、全選択/全解除ボタンを設けるなどUI配慮します。 7. **チャット画面**
メイン機能の対話UI。画面上部にタイトル「AIアシスタント」等と簡単な説明文（「過去の懸案データをもとに回答します」など）を表示。中央は対話ログエリア：ユーザーとAIの発言を吹き出しやチャットボックス風デザインで時系列に並べます。自分（ユーザー）の発言は右寄せ、AIの回答は左寄せ等で視覚的区別をつけます。各発言には時間も小さく表示。回答内に参照がある場合、それをリンクにします（例えば「懸案『サーバ起動エラー』」と出てきたらそれをクリックで詳細画面開く実装）。ログエリアは一定高さでオーバーフロー時にスクロール可能。画面下部に入力エリア：テキストボックス（多行対応、Enterで送信可）と送信ボタン。
- **回答生成中UI:** ユーザーが質問送信すると、自分の質問がログに追加され、AI側には「...(応答生成中)」のメッセージを出します。これはCSSで点滅させるか三点リーダーアニメーションなどを表示。バックエンドから回答が届いたらそれを当該メッセージに置換して表示します。もしストリーミング実装の場合は、文字が流れてくる演出も可能ですが、まずはシンプルに一括表示。 - **エラー時UI:** 何らかの理由で回答取得に失敗した場合（タイムアウト等）、AI側のメッセージとして「エラー:　回答を生成できませんでした。（原因メッセージ）」を表示します。ユーザーには再質問やリトライを促します。例えば「もう一度試す」ボタンをエラー表示内につけて、自動で同じ質問を送り直す機能があると親切です（ただし同じことを繰り返しても改善しない可能性高いので、できれば管理者に通知される仕組みが望ましい）。 - **履歴機能:** 過去のQA履歴はログエリア内にすべて残りますが、別セッション（ページ再読み込みや日をまたいだ場合）では通常消えてしまいます。そこで**履歴の永続化**を行っているため、直近の履歴を復元することも可能です。例えばユーザーごとに最後のチャットをLocalStorageに保存し、ブラウザ再訪時にそれを表示することもできます。あるいは

履歴一覧ページ（次項）で過去の会話を選択→チャット画面でそのログを再現、なんてこともできます。ただMVPではリアルタイムのものだけで十分でしょう。 8. **チャット履歴一覧画面（任意）**
（この画面は必須ではないが要件に「履歴閲覧/エクスポート（任意）」とあったため言及）
各ユーザーのこれまでの質問履歴を一覧化する画面です。例えば日時順にQの冒頭部分とAの冒頭部分を並べリスト表示し、クリックすると詳細表示またはエクスポート選択。エクスポートは単一のQAをファイル保存とか、全履歴をまとめてTXT/PDFなど出力する機能を指しますが、優先度低のため詳細割愛します。UI的には、実装するならチャット画面内に「履歴」ボタンをつけ、押すとサイドパネルに履歴一覧を出す、くらい簡易に留めたいところです。 9. **設定画面（管理者用）**
管理者だけがアクセスできる設定ページです。項目は: - 使用モデル選択: ドロップダウンまたは入力欄（モデル名を入力 or あらかじめ候補モデル名のリストを表示、例: "mistral-7b", "llama2-13b-chat" 等）。 - LLMパラメータ: スライダーや数値入力で温度や最大トークンを設定。デフォルト表示も添える。「温度0＝deterministic（決定的）、1に近いほどランダム」といった説明をTip表示。 - 検索方法切替: ラジオボタンで「キーワード検索」or「ベクトル検索（高精度）」を選択。ベクトル検索は未実装ならdisabledにしておく。 - その他設定: タイムアウト秒数（数値入力）、履歴保存件数上限（例えば各ユーザー最新100件までとか）、ログレベル選択（INFO/DEBUG）など運用向けパラメータ。 - ユーザー管理リンク: 管理者がユーザーを追加できるUIへの導線。 設定画面はフォーム送信で `/api/admin/config` にPUTリクエストを送り、成功したら「保存しました」と通知（Toastメッセージ）を出します。重大な変更（モデル変更）は再起動が必要な場合もあるため、その場合は「変更を適用するにはサーバー再起動が必要です」などメッセージを表示します。 10. **ユーザー管理画面（管理者用、必要時）**
シンプルなCRUD画面。ユーザー一覧（名前、ロール、最終ログイン日時など）と、新規登録フォーム、削除ボタン等。一般的なユーザー管理UIなので詳細は割愛します。

画面遷移図: 以上の画面を踏まえた遷移関係をMermaidの状態図で示します。

```
flowchart LR
    Login --> Main
    subgraph Main Page
        direction TB
        MainMenu(Menu) -- 懸案一覧 --> IssueList
        MainMenu -- チャット --> Chat
        MainMenu -- 設定（管理） --> Settings
    end
    IssueList -- クリック:懸案項目 --> IssueDetail
    IssueList -- 新規登録ボタン --> IssueFormNew
    IssueList -- インポートボタン --> ImportUpload
    IssueDetail -- 編集ボタン --> IssueFormEdit
    IssueDetail -- 一覧に戻る --> IssueList
    IssueFormNew -- 保存 --> IssueDetail
    IssueFormNew -- キャンセル --> IssueList
    IssueFormEdit -- 保存 --> IssueDetail
    IssueFormEdit -- キャンセル --> IssueDetail
    ImportUpload -- ファイル選択 --> ImportPreview
    ImportPreview -- 登録実行 --> IssueList
    ImportPreview -- キャンセル --> IssueList
    Settings -- ユーザー管理 --> UserMgmt
    Settings -- 戻る --> MainMenu
    UserMgmt -- 戻る --> Settings
    Chat -- 履歴一覧(任意) --> ChatHistoryList
    ChatHistoryList -- 選択 --> Chat
    ChatHistoryList -- 戻る --> Chat
```

（図: 四角は各画面コンポーネント、矢印は主要な遷移やモーダル開閉を示す。）

UI要素のデザイン詳細: - **ナビゲーションバー:** 画面上部に固定配置。アプリ名（例: "Local AI ChatTool"）のロゴ/テキスト、右側にメニューリンク（「懸案」「チャット」「設定」）。ログインしている場合はユーザー名表示とログアウトボタン。スマホ対応するならハンバーガーメニューに格納。 - **一覧画面（テーブル）:** 並べ替え機能（列ヘッダークリックでソート）やページネーション（下部にページ番号）を実装するとUXが向上。ただし1000件規模なら全件ロードしてフロントJS側でフィルタ・ソートする手もあります。初心者的にはライブラリ（React Table等）を使って簡単に済ませるのが良さそうです。 - **モーダルダイアログ:** 画像の拡大表示や削除確認など、ポップアップが必要な箇所があります。ReactであればModalコンポーネントを使用し、重ねて表示します。例えば添付画像サムネイルをクリック→モーダルに大きな画像とキャプション表示、右上に閉じるボタン。 - **フォームのUX:** 入力支援として、例えばタイトル入力欄では既存の類似タイトル候補を表示するなど高度なUXも考えられます（過去の類似懸案がないかチェックする機能）。ただ実装は複雑なのでまずはシンプルに入力のみ。代わりに、インポート時には重複チェックくらいはしても良いでしょう（同じタイトルが既に存在する場合に警告）。 - **エラーメッセージ表示:** バリデーションエラーは各フォームフィールド直下に小さな赤字テキストで表示。APIエラー（例えば保存時500エラー）は画面上部に全体メッセージで表示するか、Toast通知（右上にフラッシュメッセージ）を出すと分かりやすいです。 - **ロード中状態:** API呼び出し中は対象エリアにスピナーを表示します。例えば一覧取得中はテーブルをグレーアウト＋中央にロード中アイコン、チャット回答中は前述のtyping表示など。React Query等を使うとローディング状態管理が簡単です。初心者でも「isLoading」stateを自前管理すればOKです。

ファイル取り込みプレビューUI具体例:
取り込みプレビュー画面では、抽出結果をカードごとに表示すると述べましたが、ここで簡単な例を示します。例えばExcelから2件抽出できた場合:

```
------------------------------
| [懸案1] タイトル: [テキスト入力]          |
| 詳細: [テキストエリア]                  |
| 対策: [テキストエリア]                  |
| [ ] 登録する     [原文を見る]          |
------------------------------
| [懸案2] タイトル: [テキスト入力]          |
| 詳細: [テキストエリア]                  |
| 対策: [テキストエリア]                  |
| [x] 登録する     [原文を見る]          |
------------------------------
         (キャンセル) (登録実行)
```

各カードにはチェックボックス（デフォルトON）と、原文リンクがあります。原文リンクは、例えばExcelならセル内容一覧、Word/PDFなら該当テキスト断片を表示するモーダルを想定していますが、MVPでは省いてもいいです。ユーザーはテキストエリア内を自由編集できます。Enterで改行可能。もし長文ならテキストエリアを高さ調整可能（ドラッグでリサイズできるようにCSS設定）にする配慮も考えられます。

画像プレビュー/拡大: 懸案詳細画面で画像サムネイル（小さくリサイズしたものかCSSで縮小表示）を並べます。ライトボックス風に、クリック時に背景半透明のモーダル全面表示し、その中に<img>タグでフルサイズ画像を表示。複数ある場合は左右ナビゲーションボタンで次の画像に切り替えられるとなお良いですが、まずは一枚ずつ閉じて開き直す方式でも可。画像にはキャプション文字列も下部に表示します。

チャット画面の引用表示: 回答にreferences情報がある場合、それをUIで示します。案としては、回答吹き出しの下に小さく「参考: 'サーバ起動エラー'(懸案ID 120)」のようなリンク一覧を出すことです。複数あればカンマ区切り。クリックすると別タブまたはモーダルで懸案詳細を開けます。これでユーザーは元情報を確認

できます。将来的に、回答内に引用符号(例えば[1][2])を入れて、回答テキストと資料対応付けする高度な表示も可能ですが、MVPでは上記シンプル方式で十分でしょう。

9. 非機能要件

パフォーマンス:
- 同時ユーザー100人規模: 前述の通り、ボトルネックはLLM推論です。一問に数秒〜十数秒かかるため、例えば10人が同時に質問しただけでもレスポンス待ち行列が発生します。本システムでは**キューイング**で同時要求を直列化し、各要求の待ち時間を多少伸ばしても確実に処理する方針を取ります。もしハード的に余裕があれば、複数並列推論も検討します（例えば2つのモデルインスタンスを起動し、奇数要求はインスタンスA、偶数はBに送るなど）。ただ7Bモデルでも並列動作にはメモリ倍増が必要なので、基本は1並列とします。
- 応答時間: ユーザーがストレスなく使える応答時間は理想的には5秒以内ですが、現実にはモデル性能次第で10秒前後かかることもあります。非機能要件として**平均応答8秒以内、最悪でも20秒以内**程度を目標にします。20秒を超えたらタイムアウトエラー扱いで返答し、ユーザーには遅延を伝えます。パフォーマンスチューニングとして、プロンプトに入れるテキスト量を絞る、モデルをより軽量化（4-bit量子化版を使う等）する、または必要なら**回答候補を一旦出して返しておき後から精度高い回答に差し替える**といったUX上の工夫も考えられます（ただし後者は一貫性に問題出るのでやらない予定）。 - DBおよびAPI: SQLiteは1クエリ当たり数ms程度で結果が出る規模なので問題ありません。検索でも1000件規模なら即時応答です。APIサーバ（FastAPI/Node）は100並行リクエスト程度軽く捌けます。必要に応じてUvicornのワーカー数を増やしたり、Nodeならclusterモジュールでプロセスフォークすることもできますが、そこまでしなくても大丈夫でしょう。

セキュリティ:
- データ漏洩対策: ローカル環境で完結しているため外部へのデータ送信はありません [3] 。ネットワーク的には社内LAN内のみアクセス可能にし、ファイアウォールで外部遮断すれば機密性は確保されます。ただしブラウザ⇔サーバ間通信はHTTPになるので、環境によってはHTTPS設定することも検討します（自己署名証明書で構わないので暗号化することで盗聴対策）。
- 認可: 管理者APIは必ず管理者権限チェックを行います。実装上、リクエスト時にセッションのユーザー情報からroleを参照し、管理者でなければ403 Forbiddenを返します。フロント側でも管理者でなければ設定画面や削除ボタン等を隠す措置をしますが、直接API呼び出しされる可能性もあるためサーバ側チェックは必須です。 - 入力バリデーション: ユーザーからの入力（フォームテキスト、ファイル名、チャット質問など）は全てサニタイズ・バリデーションを行います。具体的には、SQLインジェクション対策はプリペアドステートメント利用で担保（SQLiteでのパラメータバインド）。XSS対策として、懸案内容やチャット質問/回答はエスケープして表示します。Reactは基本的にJSX内での変数展開はエスケープされるので、生HTMLを表示しない限り安全です。ただ、引用表示などでHTMLタグを差し込みたい場合（FTSのsnippet結果をそのまま表示等）は `dangerouslySetInnerHTML` を使わないよう注意するか、使う場合は信頼できる内容のみとします。 - ファイルアップロード対策: アップロード可能な拡張子は限定します。実装上はContent-Typeとファイル名でチェックし、Word/Excel/PDF以外弾く、画像もpng/jpgのみ受け付けます。さらにファイル名からのディレクトリトラバーサル攻撃（例えば "../../evil.exe" のような名前で保存先をずらす）を防ぐため、サーバ側で保存ファイル名は再構成（前述ユニーク命名）します。【パス・拡張子チェック】は必須です。加えて、画像の場合ウイルスエンジンでスキャンできるならした方がいいですが、オフライン環境では難しいため、信頼できる社内資料のみという前提で省略します。 - LLMの出力検閲: 一般的にAIの出力が不適切・有害でないかのフィルタリングもセキュリティ/リスク管理上考えます。ただ本システムの知識範囲は社内懸案であり、公序良俗に反する内容は想定しにくいです。よって特別なフィルタリング処理は入れません。ただしユーザーからの質問によりモデルが個人名等を回答に含む場合は、その個人情報が社内とはいえ慎重に扱うべきです。内部情報がモデルから漏れる恐れがある場合（例えば社の機密コードが懸案に含まれており回答で丸ごと出てくる）は、システム使用者のリテラシーに委ねます（社外秘情報をさらに外へ出さないよう注意する等）。

ログ設計:
- アクセスログ: 誰がいつどのAPIにアクセスしたかの記録。サーバサイドでミドルウェア等を使ってログファ

イルに出力します。各リクエストのメソッド、URL、ユーザーID、ステータスコード、所要時間を残し、問題発生時の調査に役立てます。100ユーザー程度なのでログ量は多くありません（1日数千行程度）。 - アプリケーションログ: サーバ内で重要イベント（ユーザーログイン成功/失敗、モデル変更、エラー発生など）をINFO以上で残します。特にエラーはスタックトレースも含めERRORログに出力し、あわせてフロントに通報する仕組みもあると便利です（例: エラー検知したら管理者メール送信や画面表示）。AI回答関連では、**質問と生成回答もログ**として残せるようにします。機密性から言えばDBのChatHistoryに残っているので二重記録は不要ですが、テキストファイルとして出力しておけばgrep等で分析しやすい利点があります。ただし回答内容に個人情報等含む可能性もあるので、扱いは社内ポリシーに従います。 - LLM応答時間モニタ: 応答時間やトークン長をメトリクスとして測定しログに出すことで、モデル性能のモニタリングをします。具体的には `/api/chat/question` 処理時間を測り、「QuestionID 77: LLM response in 8.5 sec, prompt_tokens=512, answer_tokens=128」というログをDEBUGレベルで出力します。これを蓄積し、モデル変更前後で比較したり、突然遅くなった場合に検知するのに役立てます。 - フロントログ: ブラウザ側でも想定外のエラー（JavaScript例外等）が起きたらconsole.errorや外部ロギングはできないので、ユーザーに「画面を再読み込みしてください」など案内する程度です。重要なのはバックエンドのログなので、そこを充実させます。

バックアップ:
- SQLiteのバックアップ: データベースはファイル（例: `data/database.sqlite`）であるため、そのコピーを定期取得します。シンプルにWindowsのタスクスケジューラ等で深夜にサービスを停止→DBファイルコピー→再開でも良いですし、SQLiteはオンラインバックアップAPIもあるのでシステム稼働中にバックアップを取得可能です。初心者向けには**運用ドキュメントで「アプリ停止してからDBファイルをコピーしてください」**と指示する方法が確実でしょう。バックアップ世代管理も考え、毎日のバックアップを世代別フォルダに保存し1週間分保持などを推奨します。 - 添付ファイルのバックアップ: `attachments/` ディレクトリも同様にコピーが必要です。DBと同期性を保つため、DBと同時にフォルダ全体をまとめて圧縮するのが手っ取り早いです。例えばPowerShellスクリプトで `Stop-Service MyChatApp; Copy-Item database.sqlite backup/; Compress-Archive attachments backup/attachments.zip; Start-Service MyChatApp;` のような処理を定期実行します。ファイル数1000件程度なら圧縮もすぐ終わるでしょう。なお、**インポート元ファイル**はDBに保持していないため、ユーザーがアップロードしたオリジナルファイルはこのシステムには残りません。必要ならImportedFilesテーブルからファイル名を追って手動管理できますが、基本は元ファイル保管はユーザー任せで良いでしょう。 - 設定・モデルのバックアップ: 設定値（config）はSQLiteまたは.envに入れているのでDBバックアップに含まれます。LLMモデルデータ（数GBファイル）は基本的に変更しませんが、もしローカルにしかない貴重なモデルならそちらも外部ストレージにコピーしておきます（ただし大容量なので更新時のみ手動で良いでしょう）。 - リストア手順: 万一サーバ故障等でリストアする場合は、新環境にアプリをセットアップし、バックアップしたDBとattachmentsを所定位置に戻せば復旧できます。サービス停止中であればSQLiteなので整合性問題も起きません。

運用・保守:
- モデル更新: 新しい優秀なモデルが公開された場合、管理者はモデルファイルを入手しOllamaにインストール（`ollama pull` や、ggmlファイル配置など）します。そして設定画面でモデル名を変更→保存するだけで、次の質問から新モデルが使われます。モデル更新時は一度旧モデルとの回答比較テストを行い、問題なければ本番適用とします。モデルによってはプロンプト文言の調整（例えば日付形式や禁止事項記述）が必要なので、その点も検証します。 - プロンプト調整: 利用が進むと、モデルが期待通り回答しないケースや、余計な一言を付ける等の癖が見えてくるでしょう。その際はSystemプロンプトに追記する、回答フォーマット指定を加えるなど微調整を行います。これも設定で変更可能にする案もありますが、悪用されると困るのでコード内定数で持ち、バージョンアップ時に変更する形でも構いません。 - オフライン運用: 完全にインターネットに繋がらない環境でも動作するよう、前提となるソフトウェア（Node.js/Python, Ollama, モデルファイルなど）は事前に用意します。Windows版Ollamaがもし未提供ならWSL2かDockerでLinux環境を作ります。WSLに抵抗ある場合は、LocalAI等Ollama互換のWindows実行可能サーバを使用することも検討します[12]。いずれにせよ、一度構築してしまえばオフラインで継続利用できます。アップデート（例えばセキュリティ修正）時のみ一時的なネット接続が必要かもしれません。 - モデルライセンスと社内利用: オープンモデルとはいえ商用利用可否や再配布禁止などライセンスが様々です。社内利用であってもコンプライアンス上確認が必要です。例えばLlama2は商用利用可ですが、ELYZAのモデルは商用利用の場合契約が必要、といった

ケースもあります。運用前に選定モデルのライセンス条項を確認し、問題ないものを使います。モデルを差し替える場合も同様です。 - 障害対応: 本システムはシンプル構成ゆえ、起こり得る障害は「アプリが落ちた/応答しない」「モデルがロードできない」「ストレージ満杯による保存失敗」などです。アプリ停止はWindowsサービスとして登録し自動再起動設定すればある程度カバーできます。モデルエラーはログで原因を追求し、例えばメモリ不足ならモデルサイズ縮小や起動パラメータ変更で対応します。ストレージは添付画像が増えると圧迫しますが1000件程度では高が知れているので、問題になりにくいでしょう。年単位で見るとログファイル肥大化もあるので、ログローテーション設定をしておきます。

10. 実装計画（初心者向け）

初心者がスムーズに開発・導入できるよう、段階的な実装と環境準備の計画を立てます。

段階的リリース案: 1. **MVP (Minimum Viable Product) リリース:**
最小機能をまず作ります。これは**手動懸案登録・検索 + チャット回答**の実現を目標とします。具体的には、ユーザーがフォームから懸案を登録でき、一覧・詳細を閲覧でき、チャット画面で質問するとその懸案データから回答が返ってくる、という一連の流れです。ファイルインポートや画像添付はこの段階では**含みません**。認証も簡略化し、シングルユーザー前提または仮の管理者認証だけで済ませます。まずこのMVPで、**RAGによる回答生成**が期待通り動くかを検証します。不具合修正やUI改善もこのフェーズで繰り返します。MVPが動けば、それ自体がポートフォリオとして十分インパクトがあります（社内の非効率をAIで解決する具体例として示せる）。 2. **第二段階: ファイル取り込み機能追加**
MVP後、懸案データを効率よく蓄積するためのファイルインポート機能を実装します。まずExcelのテンプレート取り込みから着手すると理解しやすいでしょう。決まった列のExcelを読み込み→複数懸案登録までを作り、その後Word/PDFにも対応を広げます。抽出ライブラリの扱いやプレビューUIなどMVPより難易度高い部分ですが、1段階目で基盤（API通信やDB操作）ができているので、それを応用する形です。取り込み機能が完成すれば、既存のバラバラな知見を一気にシステムに蓄積でき、**懸案データベースの充実**が図れます。 3. **第三段階: 画像添付機能追加**
次に画像アップロードと表示の機能を加えます。フロントエンドでのファイル選択、プレビュー表示、バックエンドでの受け取り・保存処理、など一通りのフローを実装します。この段階ではOCRは入れずキャプションのみ活用します。画像表示ができればUIがリッチになり、**ユーザビリティの向上**につながります。Reactのfile inputや<img>表示の基礎的扱いも習得できます。 4. **第四段階: RAG強化**
システムとして基礎機能が揃った後、AI回答の精度向上に取り組みます。具体的には、**ベクトル検索の導入**や**プロンプトのチューニング**です。LangChain等を組み込んでみてもいいでしょう。ここは実装より実験的な作業が増えますので、時間に余裕があれば挑戦します。特にベクトル検索（FAISS/Chroma）は、Pythonであればインストールして数十行書けば動かせるため、初心者がAIらしい高度な仕組みに触れる良い機会になります。これが成功すれば**ポートフォリオのアピールポイント**としても「ベクトルDBを用いた高度な検索を実装」と書けるので効果的です。

各段階で小さくリリース（もしくはデモ）し、フィードバックを得て改善する形が望ましいです。例えばMVP完成時点で社内の想定ユーザーに試してもらい、UIの不備や回答の質を確認します。その後段階2,3で追加機能を実装しつつバグ修正と微調整を続けます。段階4はあれば尚良し程度なので、ポートフォリオ提出期限と相談して無理のない範囲で進めます。

最初に作る最小機能セット（MVP）:
再度MVPの中身を整理すると: - ユーザー登録（省略可能なら固定ユーザーで開始） - 懸案登録（フォームでtitle/description/resolution、DB保存） - 懸案一覧・詳細閲覧（DBから取得表示、検索は単純部分一致でもOK） - チャット画面（質問送信→バックエンドで全文検索&LLM応答→表示） - 管理者のモデル名設定（なくてもデフォルトモデルでOKなので後回し） - ログイン認証（まずはハードコードユーザー or ノーログインでOK）

これだけ実装してまず**システムが一通り動く**ことを目指します。UIは多少不格好でも機能することを優先し、リファクタやデザイン調整は後でまとめて行います。

開発環境セットアップ手順（Windows）：
初心者向けに、Windows10/11上での環境構築手順をまとめます。

- **1) 基本ツールのインストール:**
- **Git:** ソース管理用にGitをインストール（https://git-scm.com）。GUIクライアントでもOK。
- **Node.js:** フロントエンドビルドやパッケージ管理に必要。https://nodejs.org から最新版LTSを導入。これでnpmも使えるようになる。
- **Python:** バックエンドがPythonの場合、3.10以降をインストール（https://www.python.org）。インストーラで「Add to PATH」にチェックを忘れずに。pipも付属する。
- **VSCode:** 開発用エディタ。https://code.visualstudio.com から入手。インストール後、Python拡張やESLint拡張などを入れると便利。

- （**.NET SDK:** もしC#案ならhttps://dotnet.microsoft.com/downloadから .NET 7以上 SDKを入れる。ただ今回はPython採用なので不要。）

- **2) Ollama導入 or 代替:**
OllamaはMac/Linux向けのLLMサーバなので、Windowsでは直接動きません。代替策:

- **WSL2 + Docker:** Windows上でLinux環境を動かす。WSL2を有効化し、Ubuntu等をインストール。さらにDocker Desktopを入れてWSL連携する。これでLinuxコンテナ（Ollama公式イメージ）を動かせます。例えばPowerShellから `docker run -p 11434:11434 ghcr.io/jmorganca/ollama` などで起動可能 [4]。
- **LocalAI:** Windows用のローカルLLMサーバ。https://github.com/go-skynet/LocalAI で提供されており、OllamaのAPI互換モードもあります [12]。これを利用すると、Ollama用に書いたコードをそのまま使える可能性があります。Go製なのでexeがあります。
- **直接Pythonでモデルロード:** TransformersやSentencePieceを使い、モデルファイル（GGMLやPT）を直接読み込んで推論する方法。ただし7Bでもメモリ食うためあまり現実的ではないかも。初期はこれでも試せるが、後でOllama/LocalAIに切り替える前提で簡易実装に留めるならありです。

最も簡単なのはLocalAIを使うことです。LocalAIのリリースからWindows用binaryをダウンロードし、`localai.exe` を起動するだけでREST APIが立ち上がります。事前にモデルの準備が必要なので、例えば**Mistral 7B**を使うなら、HuggingFaceからggmlモデルをダウンロードし（`mistral-7b-instruct-v0.1.Q4_0.bin` 等）、LocalAIのmodelsフォルダに配置します。起動後、`http://localhost:8080/v1/models` でモデル一覧を確認し、HTTP APIで推論できます。LocalAIはOpenAI API互換のエンドポイントを持つため、コード側もOpenAI API呼び出し形式にすれば簡単に試せます。

今回は便宜上「OllamaのAPIを呼ぶ」体裁で記述しましたが、WindowsではLocalAI (OpenAI APIモード)に差し替える想定です。その場合、バックエンドでの呼び出しはPythonならopenaiパッケージを使い、`openai.ChatCompletion.create(model="mistral-7b", messages=[...])` で済みます。モデル名はLocalAI上に登録したものを使います。**設計上の仮定**として、環境構築時に適切なLLMサーバを用意できる前提で進めています。

- **3) プロジェクトディレクトリ構成:**
Gitで新しいレポジトリを作成し、以下のような構成で進めます。

```
project-root/
├── backend/
│   ├── app.py            # FastAPIメイン
```

```
│   ├── requirements.txt   # Python依存パッケージ
│   ├── models/            #  （AIモデルファイル置き場ではない、DBモデルやPydanticモデル用フォルダ想定）
│   ├── services/          # RAGやファイル処理等のサービスロジック
│   ├── static/            # 静的ファイル（添付画像提供のための空フォルダ）
│   └── ... (other modules)
├── frontend/
│   ├── package.json       # Node依存パッケージ（React, etc）
│   ├── public/            # publicディレクトリ(index.htmlなど)
│   └── src/               # Reactソース（JSX/TSX, CSS）
│       ├── components/    # 再利用コンポーネント
│       ├── pages/         # 各ページ（React Router用）
│       └── App.js / App.tsx  # エントリポイント
├── data/
│   ├── database.sqlite    # SQLite本体（初回実行時に生成）
│   └── attachments/       # 添付ファイル保存フォルダ
├── .env                   # 環境変数定義（DBパス，LLMモデル名，etc）
└── README.md              # セットアップ・起動手順記載
```

- backendとfrontendは別々に起動（devモードではport違い）し、プロダクション時にはビルドしたfrontendをbackendのstaticとして配信する構成も可能です。
- Pythonの仮想環境(venv)を作成し、その中にFastAPI, SQLite, openai, langchain等の必要ライブラリをpip installします（requirements.txtを用意しpip install -rする）。
- ReactはCRA(Create React App)やViteで初期プロジェクトを作成し、必要なUIライブラリをnpmインストールします（例えば Material-UI, Axiosなど）。
- .envファイルには、例えば `MODEL_NAME=mistral-7b`, `MAX_TOKENS=1024`, `DB_PATH=../data/database.sqlite` 等設定を書きます。Python側で `python-dotenv` を使って読み込みます。初心者ならconfig.pyに直書きでもいいですが、セキュア情報がないので.envでOKでしょう。

- attachmentsフォルダはバックエンドから書き込むので、読み書き権限が必要です（Windowsでは特に問題ないが、実行ユーザー権限に注意）。

- **4) 開発の進め方:**
  まずbackendから実装するとデータ構造が固まりやすいです。Pydanticモデルを定義し、FastAPIのエンドポイントを仮実装してみて、Swagger UIで試します。DBアクセスはsqlite3かSQLAlchemyを使いますが、初心者ならシンプルにsqlite3でクエリを書く方が理解しやすいかもしれません。
  フロントはAPIができてから繋ぎ込みますが、モックデータで先にUIを作ることもできます。例えば懸案一覧はダミーデータ配列で表示しておき、後でfetchに置き換える等。

- **5) 実行・テスト:**
  開発時は、backendは `uvicorn app:app --reload` で起動し、frontendは `npm start` で起動します。クロスオリジンになるのでFastAPIにCORS許可設定を追加（`fastapi.middleware.cors import CORSMiddleware` でhttp://localhost:3000を許可など）。ReactからaxiosでAPIを呼ぶ際は `http://localhost:8000/api/issues` にアクセスすればデータが取れる状態にします。
  MVP機能実装後、手動で以下のテストを行います:

- 懸案を3件ほど登録 → 一覧に出るか、検索はヒットするか
- チャット画面で、登録した懸案に関する質問を入力 → AIがそれらしい回答を返すか（初回はモデル動作テストも兼ねている）
- 異常系: 空タイトルで登録→エラー表示、存在しないIDの懸案取得→404処理、AI質問でLLMサーバ落として応答→エラーメッセージ表示、など。

テスト方針: - ユニットテスト: 可能ならPythonのunittestやpytestで、RAG検索関数やテキスト抽出関数のテストを書きます。例えば `extract_text_from_docx()` にサンプルファイルを与えて期待通りの文字列が出るか、`search_issues("起動　エラー")` がちゃんと関連Issue返すか、といったテストです。LLM呼び出し部分は外部依存なのでモックにして、応答例を決め打ちして処理フローを確認します。 - 結合テスト: API全体のテストとして、FastAPIならTestClientを使ってエンドポイントにリクエストを送り、DB結果を検証するテストを書けます。ただ初心者には高度なので、**手動テスト＋ポストマン**などAPIクライアントでの試験をメインにします。Postmanで一連のリクエストシナリオ（ログイン→登録→取得→質問　等）をスクリプト化しておけば、リグレッションテストにも使えます。 - 負荷テスト: 100並列を想定した負荷テストは大掛かりですが、簡易にはApacheBench（ab）で `/api/chat/question` に10並列×10回=100回叩いてみるとか、JMeterでスクリプト作る方法があります。LLM部分は時間かかるので明確な数値は出ませんが、サーバがエラーを返さず処理しきれるか確認します。 - UXテスト: 社内の同僚に触ってもらい、操作上の違和感や不明点をフィードバックしてもらいます。初心者が作ると専門知識前提のUIになりがちなので、誰でも迷わず使えるかチェックします。例えば「懸案って言葉がわかりにくいから 'ナレッジ' と表示した方がいい」などの意見があれば取り入れます。

以上、設計上の判断理由:　本設計では「初心者でも実装できること」「ローカル完結によるデータ安全性」「既存資産を活かすことで効率化」という観点を重視しました。各章で提示したように、要件の一つ一つに対し現実的な技術を当てはめ、無理のないスコープでプランしています。また、過去の日記で述べられていた**現在の職場の非効率**（Excel管理の限界やナレッジ共有不足）を踏まえ、それを解決するシステムとしてポートフォリオ価値が高まるよう配慮しました。段階的実装計画により無理なく完成まで持っていける見込みです。もし不明点や前提違いが判明した場合は、**設計上の仮定**として注記した部分（例えば認証やOCR対応可否）を再調整します。

未確定事項と今後の検討:
- ログイン認証をどのタイミングで導入するか（MVPでは省略したが本番運用時は必要か）。 - WindowsでのLLM実行方式の最適解（LocalAIで問題ないか、Docker使用許可はあるかなど）。 - ファイルインポート時のフォーマット詳細（ユーザーにテンプレを周知する必要があるので、運用部門との調整が必要）。 - ベクトル検索導入時の追加リソース（ベクトルDBのサービスを別プロセスで立てる場合、運用コスト増になるので、社内承認が必要かも）。 - モデルのチューニングにどこまで踏み込むか（社内データで追加トレーニングは今回はしない予定だが、要望次第ではLoRA調整もあり得る）。

これらはプロジェクト進行中に適宜クリアにしつつ、必要なら設計変更を行います。現時点の計画に大きな齟齬はないと考えていますので、この設計書を叩き台に開発を進めたいと思います。

---

1 2 Excelで情報共有は古い？ナレッジマネジメントツールと比較
https://boxil.jp/mag/a8594/

3 13 Run Mistral 7B Locally: Step by Step Guide with Intro to Agents | by Vadim Korotkikh | Medium
https://medium.com/@vadimkorotkikh/local-mistral-7b-quick-start-guide-c6d8326f494f

4 5 10 Implement a simple RAG with Ollama, Qdrant and LangChain | by Filip | Medium
https://medium.com/@filipzupancic123/implement-a-simple-rag-with-ollama-qdrant-and-langchain-cc47beabc290

6 SQLite FTS5 Extension
https://www.sqlite.org/fts5.html

7 node.js module for extracting text from html, pdf, doc, docx ... - GitHub
https://github.com/devmehq/extract-text

8 Extract Text from Images, PDFs, Docs, XLSs, and more in Python
https://davidemanske.com/extract-text-from-images-pdfs-docs-xlss-and-more-in-python/

9 officeparser - NPM
https://www.npmjs.com/package/officeparser

11 Local RAG with Ollama, LiteLLM, and Qdrant - Gabriel Mongeon
https://gabrielmongeon.ca/en/2025/12/local-rag-ollama-litellm-qdrant/

12 How to Run a Local LLM via LocalAI, an Open Source Project
https://thenewstack.io/how-to-run-a-local-llm-via-localai-an-open-source-project/

14 Top 3 Recommended Japanese LLMs｜Thorough Comparison of Large-Scale Language Models Specialized for Japan [Latest 2025] | AI & Annotation Blog | Human Science Inc.
https://www.science.co.jp/en/annotation_blog/39133/