実務レベル習得のための技術習得ロードマップ (27卒向け)

1. プロジェクト概要

本ロードマップでは、個別の技術を断片的に学ぶのではなく、全ての技術スタックを統合した一つのアプリケーションを作成することで、各技術の相互作用と「なぜその技術が必要なのか」を深く理解することを目的とします。

1.1 開発ターゲット: DocuBrain-Agent

社内ドキュメント（仕様書や議事録）をAIが解析し、対話形式で回答するRAG（検索拡張生成）アプリケーション。さらに、MCP（Model Context Protocol）を用いて、AIが外部ツールを操作する拡張性を持たせます。

1.2 習得する技術スタック

- **Backend:** Python (FastAPI), Pydantic
- **Frontend:** TypeScript (React/Vite)
- **Infra/Cloud:** Docker, Google Cloud Run, Firebase (Auth/Hosting)
- **AI/Data:** OpenAI API / Gemini API, Vector DB (ChromaDB/Qdrant), MCP
- **Methodology:** Agile, Git Flow

2. 技術ごとの学習ゴール（面接対策キーワード）

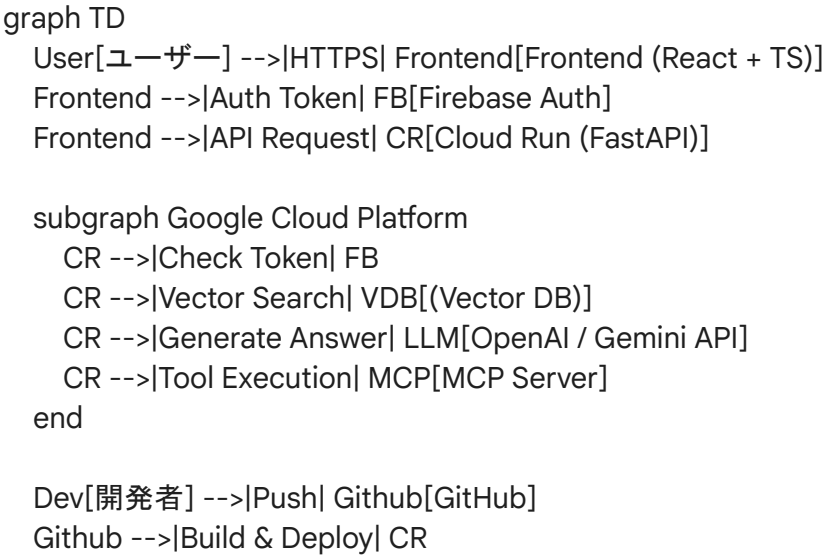
面接官に「実務で通用する基礎がある」と判断させるための、各技術の理解度定義です。

技術カテゴリ	具体的な技術	面接で語れるべきキーワード・概念	学習のポイント
Backend	**Python (FastAPI)**	非同期処理 **(Async/Await)** Dependency Injection (依存性注入) Pydanticによる型定義・バリデーション ASGIサーバー (Uvicorn)	Django/Flaskとの違い（パフォーマンスとモダンな型安全性）を理解し、Swagger UIを用いたAPI設計ファーストの開発フローを説明できること。
Frontend	**TypeScript**	静的型付けの恩恵 Interface vs Type Generics (ジェネリクス) Union Types / Utility Types	any型を使わずに実装することの重要性（保守性と安全性）。バックエンドの型定義との整合性をどう保つか。
Infra	**Docker**	コンテナのライフサイクル Dockerfile vs docker-compose マルチステージビルド Volumeによるデータ永続化	「私の環境では動きました」問題を解決するための技術であることを理解する。イメージサイズ軽量化の工夫。

Cloud	**GCP (Cloud Run)**	サーバーレスコンテナ ステートレスアーキテクチャ コールドスタート IAM (Identity and Access Management)	従来のVM（EC2/GCE）と比較した際の、運用コスト削減とスケーラビリティのメリット。
AI / Data	**LLM / Vector DB**	**RAG (Retrieval-Augmented Generation)** Embeddings (ベクトル化) Cosine Similarity (コサイン類似度) Hallucination (幻覚)	LLMの知識の限界（学習カットオフ）をRAGでどう補完するか。トークン制限への対処法。
Advanced	**MCP**	**Model Context Protocol** Client-Host-Server モデル	AIとデータソース/ツールを接続するための標準規格としてのMCPの意義。

3. システムアーキテクチャ設計図案

開発するアプリの全体像です。



4. 週次アクションプラン（詳細版）

Week 1: バックエンドの堅牢化とコンテナ化

テーマ:「環境に依存しない、堅牢なAPIサーバーの構築」

- **Day 1: FastAPIセットアップと型定義**
  - poetry または pip で環境構築。
  - FastAPIをインストールし、GET /health エンドポイントを作成。
  - 重要: Pydantic モデルを作成し、リクエストのバリデーションを実装する。
- **Day 2:** ビジネスロジックの実装
  - ドキュメントのアップロード用エンドポイント (POST /upload) を作成。
  - ファイルを受け取り、テキストを抽出する処理を実装。
- **Day 3: Docker化 (Dockerfile)**
  - python:3.11-slim をベースイメージに使用。
  - 課題: マルチステージビルドを採用し、ビルドステージと実行ステージを分けることでイメージサイズを軽量化する。
- **Day 4: Docker Compose**による構成
  - docker-compose.yml を作成。
  - APIサーバーとDB（この段階ではPostgreSQLなどを練習で入れてみる）を同時に立ち上げる。
  - ボリュームマウント設定を行い、ホスト側のコード変更がコンテナに即時反映されるようにする（ホットリロード）。
- **Day 5:** リファクタリングと**Swagger**確認
  - /docs にアクセスし、Swagger UIからAPIが正常に動作することを確認。
  - コード内のハードコーディングを排除し、.env ファイルで環境変数を管理する。

Week 2: AIロジックの実装とRAG構築

テーマ:「AIに独自の知識を与える（RAGの実装）」

- **Day 1: Vector DB**の導入
  - ChromaDB または Qdrant をDocker構成に追加。
  - Pythonクライアントを用いて、Vector DBへの接続確認を行う。
- **Day 2: Embeddings**（ベクトル化）の実装
  - OpenAI API (text-embedding-3-small) または Gemini API を契約・設定。
  - アップロードされたテキストデータを「チャンク（小分け）」に分割し、ベクトル化してDBに保存する処理を実装。
- **Day 3:** 検索ロジックの実装
  - ユーザーの質問文をベクトル化し、DBから「距離が近い（類似した）」ドキュメントを検索する関数を作成。
- **Day 4:** 生成（**Generation**）の実装
  - 検索して見つかったドキュメントを「コンテキスト（参考情報）」としてプロンプトに組み込む。
  - LLMに「以下の参考情報を元に回答してください」と指示を出す処理を実装。
- **Day 5:** 精度調整と**API**化
  - チャット用エンドポイント (POST /chat) を完成させる。
  - 回答が不自然な場合のプロンプト調整（System Promptの改善）。

Week 3: クラウドデプロイとフロントエンド接続

テーマ:「サービスとして公開し、ユーザーインターフェースを作る」

- **Day 1: GCP Cloud Run**へのデプロイ
  - Google Cloud SDK (gcloud) のセットアップ。
  - Artifact RegistryにDockerイメージをPush。
  - Cloud Runサービスを作成し、デプロイ。環境変数をCloud Console上で設定。
- **Day 2: Frontend**初期構築 **(TypeScript)**
  - npm create vite@latest frontend -- --template react-ts
  - Tailwind CSSなどのUIフレームワークを導入（見た目を整えるコストを下げる）。
- **Day 3: API**連携と型安全性
  - fetch または axios でCloud Run上のAPIを叩く。
  - 重要: バックエンドのPydanticモデルに対応するTypeScriptの interface を作成する（types.ts）。ここがズレるとバグの原因になることを体験する。
- **Day 4: Firebase Auth**導入
  - Firebaseコンソールでプロジェクト作成。
  - Frontendにログイン画面（Googleログイン等）を実装。
  - Backend側で、HTTPヘッダーのBearerトークンを検証する依存関数 (Depends) を実装。
- **Day 5:** フロントエンドのデプロイ
  - firebase deploy コマンドで、FrontendをFirebase Hostingに公開。
  - Cloud RunとFirebase Hosting間のCORS（Cross-Origin Resource Sharing）設定を行う。

Week 4: 発展技術とMCP、アジャイル開発の整理

テーマ:「最新技術への挑戦と、エンジニアとしての振る舞い」

- **Day 1: MCP (Model Context Protocol)** の調査と実験
  - MCPの公式ドキュメントを読み、Core Concept（Client, Host, Server）を理解する。
  - 「現在の時刻を返す」「特定のWebサイトを検索する」といった単純なToolを持つMCPサーバーをPythonで書く。
- **Day 2:** エージェント機能の統合
  - RAGで答えられない質問が来た場合、MCPツールを使って情報を探しに行くロジック（簡易版）を検討する。
  - または、GeminiのFunction Calling機能を試してみる（MCPの概念理解に役立つ）。
- **Day 3: Git**運用と**CI/CD**の真似事
  - ブランチ運用 (feature/add-mcp-support など) を徹底する。
  - GitHub Actionsのymlファイルを書き、Push時にLint（コード解析）やTestが走るように設定してみる（動かなくても設定しようとした痕跡が大事）。
- **Day 4:** ポートフォリオとしてのドキュメント作成 **(README.md)**
  - 構成案:
    1. アプリ名と概要
    2. 使用技術スタック（バージョン含む）
    3. アーキテクチャ図（Mermaidや画像）
    4. こだわった技術ポイント（苦労話） ← ここが面接で一番効く
    5. セットアップ方法 (docker-compose up で動くように)
- **Day 5:** 模擬面接練習
  - 作ったアプリを見ながら、「なぜFastAPIを選んだのか？」「RAGの精度向上で何をしたか？」を自問自答し、回答を用意する。

5. ハンズオン課題用：ディレクトリ構成案

開発を始める際、以下の構成を参考にしてください。

```
docubrain-agent/
├── backend/              # FastAPI Project
│   ├── app/
│   │   ├── main.py       # Entry point
│   │   ├── api/          # API Endpoints
│   │   ├── core/         # Config (Env vars)
│   │   ├── db/           # Vector DB connection
│   │   ├── models/       # Pydantic Models
│   │   ├── services/     # Logic (RAG, MCP)
│   │   └── utils/
│   ├── Dockerfile
│   ├── pyproject.toml    # Poetry dependencies
│   └── requirements.txt
├── frontend/             # React + TS Project
│   ├── src/
│   │   ├── components/
│   │   ├── hooks/
│   │   ├── types/        # TS Interfaces
│   │   └── App.tsx
│   ├── Dockerfile        # (Optional for dev)
│   └── package.json
├── docker-compose.yml    # Local development orchestration
└── README.md
```