Ｄｉｆｙによるエンタープライズ ＡＩ設計

第一版

セキュアな企業ＡＩ基盤の構築と運用

Gen K. Mariendorf

まえがき

このコンテンツは私自身のＤｉｆｙの理解のために書き下したものである。最初はローカル LLM（DeepSeek）と RAG を用いた業務利用可能な AI 基盤の構築・検証プロジェクトを始めて、PoC → 設計 → 検証 → AWS 展開までを再現するための技術実践書になる予定である。

以下がこの本の基本構成である。

- 動かす（試行編）
  - 第１章　Ｄｉｆｙローカル環境の構築
  - 第２章　ＲＡＧの実装と検証
- 理解する（構造編）
  - 第３章　ＲＡＧの理論構造と実践理解
  - 第４章　Ｄｉｆｙ内部構造の理解
- 使う（実務編１）
  - 第５章　実務導入パターン
- 開発し評価する（実務編２）
  - 第６章　開発スタイル
- セキュリティ
  - 第７章　セキュリティとガバナンス
- 将来展望
  - 第８章　将来展望

目次

第 1 章

Ｄｉｆｙローカル環境の構築

この章では難しい理論やシステム説明は無しで、早速Ｄｉｆｙを動かしてみましょう。簡単な「具体的スクリプト」と方法をこの章ではまとめます。前提条件は OS は Linux、Dify は Docker Compose、Ollama は Docker ホスト側で動かす、モデルは **deepseek-r1:1.5b** を使用します。

ゴール

- Ollama + `deepseek-r1:1.5b` 起動
- Dify（Docker）起動
- **Dify から Ollama に接続して、チャット応答が返る**

要するに、Dify（Docker 上で動作）→ Ollama [Deepseek-r1:1.5b] と繋いで動かしてみる訳です。

1.1 Ollama 導入 & deepseek-r1:1.5b の起動

Ollama インストール

```
$ curl -fsSL https://ollama.com/install.sh | sh
```

モデル取得＆単体テスト

```
$ ollama pull deepseek-r1:1.5b
$ ln -vs /usr/share/ollama/.ollama $HOME
$ ollama run deepseek-r1:1.5b
```

モデルの保存場所は通常以下になる。

この例では symlink を $HOME（ホームディレクトリ）へ作成

```
$ du -sh /usr/share/ollama/.ollama/models/
1,1G /usr/share/ollama/.ollama/models/
```

- 環境変数：モデルの保存場所変更する場合は設定可。

```
$ export OLLAMA_MODELS=/usr/share/ollama/.ollama/models/
```

DeepSeek を Ollama でダウンロード

```
$ ollama pull deepseek-r1:1.5b
pulling manifest
pulling manifest
pulling aabd4debf0c8: 100% 1.1 GB
pulling c5ad996bda6e: 100% 556 B
pulling 6e4c38e1172f: 100% 1.1 KB
pulling f4d24e9138dd: 100% 148 B
pulling a85fe2a2e58e: 100% 487 B
verifying sha256 digest
writing manifest
success
```

DeepSeek の起動：受け答えしてみる。→ 成功

```
$ ollama run deepseek-r1:1.5b
>>> What is your name?
Greetings! I'm DeepSeek-R1, an artificial intelligence assistant
created by DeepSeek. I'm at your service and would be delighted
to assist you with any
inquiries or tasks you may have.

>>> Send a message (/? for help)
```

1.2 Dify 事前チェック（Docker が動くか確認）

```
$ docker --version
$ docker compose version
```

出力例

```
$ docker --version
Docker version 28.1.1, build 4eba377

$ docker compose version
Docker Compose version v2.35.1
```

GPU コンテナを念の為チェック

`nvidia-smi`コマンドで GPU のチェックをします。

```
$ nvidia-smi
```

```
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.230.02             Driver Version: 535.230.02   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 4060 ...    On | 00000000:01:00.0 Off |                  N/A |
| N/A   35C    P0              N/A / 115W |     14MiB /  8188MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      1752      G   /usr/lib/xorg/Xorg                            4MiB |
|    0   N/A  N/A      5570      G   /usr/lib/xorg/Xorg                            4MiB |
+---------------------------------------------------------------------------------------+
```

図 1.1: nvidia-smi の結果

以下のようなシステム構成であることが分かりました。

GPU	VRAM	Driver	CUDA
GeForce RTX 4060	8GB	535.230.02	12.2

1.3 Dify（Docker Compose）を起動

公式の Docker Compose クイックスタートを参考にします。(docs.dify.ai Quick)

Dify を clone（推奨：最新リリースタグ）

```
$ sudo apt-get update
$ sudo apt-get install -y jq curl git

git clone --branch "$(curl -s https://api.github.com/repos/
langgenius/dify/releases/latest | jq -r .tag_name)" https://
github.com/langgenius/dify.git
cd dify/docker
cp .env.example .env
docker compose up -d
```

起動確認（主要コンテナから起動）

```
docker compose ps
```

起動確認：主要コンテナが稼働中

```
NAME IMAGE COMMAND SERVICE CREATED STATUS PORTS
docker-api-1 langgenius/dify-api:1.13.0 "/bin/bash /entrypoi…"
api 5 minutes ago Up 5 minutes 5001/tcp
docker-db_postgres-1 postgres:15-alpine "docker-entrypoint.s…"
db_postgres 5 minutes ago Up 5 minutes (healthy) 5432/tcp
docker-nginx-1 nginx:latest "sh -c 'cp /docker-e…" nginx 5
minutes ago Up 2 minutes 0.0.0.0:80->80/tcp, [::]:80->80/tcp,
0.0.0.0:443->443/tcp, [::]:443->443/tcp
docker-plugin_daemon-1 langgenius/dify-plugin-daemon:0.5.3-local
"/bin/bash -c /app/e…" plugin_daemon 5 minutes ago Up 5 minutes
0.0.0.0:5003->5003/tcp, [::]:5003->5003/tcp
docker-redis-1 redis:6-alpine "docker-entrypoint.s…" redis 5
minutes ago Up 5 minutes (healthy) 6379/tcp
docker-sandbox-1 langgenius/dify-sandbox:0.2.12 "/main" sandbox 5
 minutes ago Up 5 minutes (healthy)
docker-ssrf_proxy-1 ubuntu/squid:latest "sh -c 'cp /docker-e…"
ssrf_proxy 5 minutes ago Up 5 minutes 3128/tcp
docker-weaviate-1 semitechnologies/weaviate:1.27.0 "/bin/weaviate
 --hos…" weaviate 5 minutes ago Up 5 minutes
```

```
docker-web-1 langgenius/dify-web:1.13.0 "/bin/sh ./entrypoin…"
web 5 minutes ago Up 5 minutes 3000/tcp
docker-worker-1 langgenius/dify-api:1.13.0 "/bin/bash /entrypoi…
" worker 5 minutes ago Up 5 minutes 5001/tcp
docker-worker_beat-1 langgenius/dify-api:1.13.0 "/bin/bash /
entrypoi…" worker_beat 5 minutes ago Up 5 minutes 5001/tcp
```

Dify の初期セットアップ画面は通常 `http://localhost` （ポート８０番）で見えます（環境によりポートが異なる場合あり）。

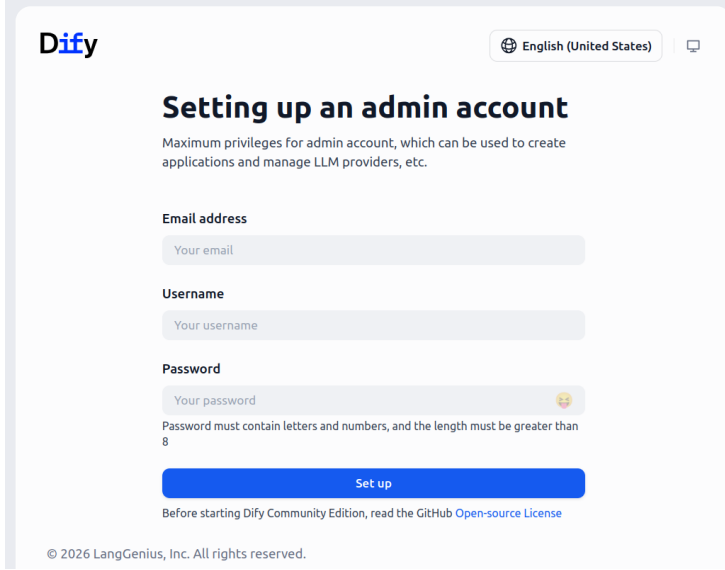

図 1.2: Dify Login

1.4 Dify → Ollama 接続

Dify 公式は「Docker で Dify を動かすなら、Ollama の URL は `host.docker.internal` か LAN IP を使う」と明記しています (legacy-docs.dify.ai Ollama)。Linux は `host.docker.internal` が効かない/不安定なことがあるので、LAN IP 方式が推奨になります。

ホスト LAN IP を取得

```
ip route get 1.1.1.1 | awk '{print $7; exit}'
```

例：192.168.1.20 が出たら、以下は http://192.168.1.20:11434

Ollama が LAN から見えるようにする（重要）

Ollama はデフォルトで 127.0.0.1:11434 にバインドします。外部（＝Dify コンテナ）から見せるには環境変数のOLLAMA_HOSTを変えます。

Ollama を LAN バインドで実行 (OLLAMA_HOST 環境変数を使うだけ)

ターミナルで：

```
$ LAN_IP=$(ip route get 1.1.1.1 | awk '{print $7; exit}')
$ OLLAMA_HOST=$LAN_IP:11434 ollama serve
```

別ターミナルで疎通確認：

```
$ curl http://$LAN_IP:11434/api/tags
```

例えば、以下のような JSON が返れば設定完了。恒久化する場合は systemd に環境変数を入れるやり方が定番ですが、最初は簡易設定で十分です。


```
{"models":[{"name":"deepseek-r1:1.5b","model":"deepseek-r1:1.5b",
"modified_at":"2026-02-13T06:56:58.294031073+01:00","size"
:1117322768,"digest":"
e0979632db5a88d1a53884cb2a941772d10ff5d055aabaa6801c4e36f3a6c2d7"
,"details":{"parent_model":"","format":"gguf","family":"qwen2","
families":["qwen2"],"parameter_size":"1.8B","quantization_level":
"Q4_K_M"}}]}
```


Dify の画面で Ollama プロバイダー設定

Dify 管理画面 → **Settings** → **Model Providers** → **Ollama**

[Add Model] ボタンを押して、

- **Model name**: deepseek-r1:1.5b → Dify 公式の DeepSeek+Ollama 事例でも同系モデルを前提
- **Base URL**: http://<LAN_IP>:11434 → もしくは http://host.docker.internal:11434 が効く環境ならそれも可能
- Model Type: Chat

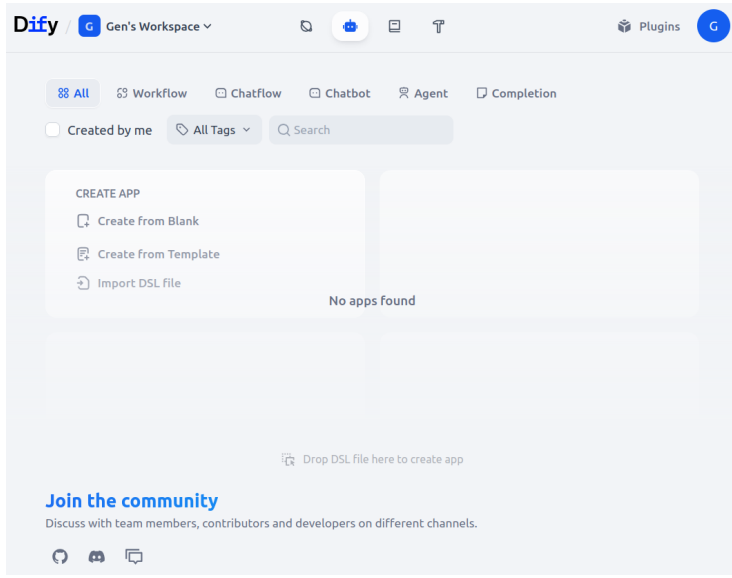

図 1.3: Dify ログイン画面

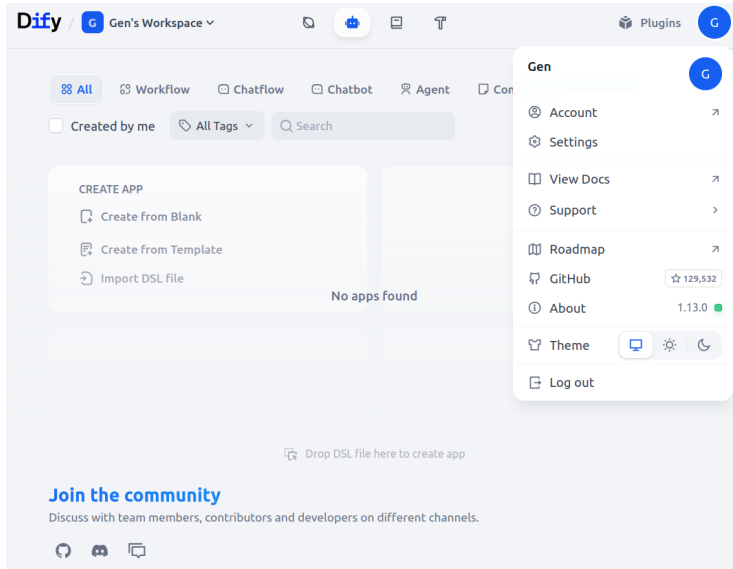

図 1.4: Dify 管理画面

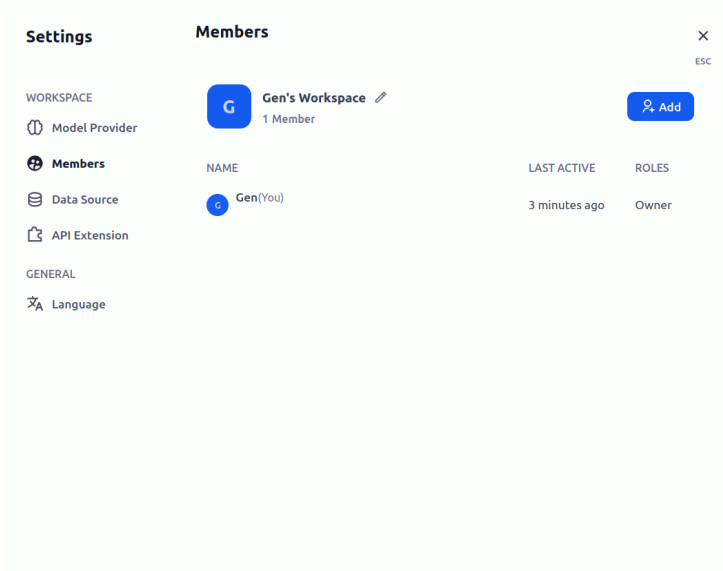

図 1.5: Dify 設定画面

- Context length: 4096（不明ならデフォルト）

[Add] ボタン追加したら成功。

最小チャットアプリを作って動作確認

Dify で新規 App → Chat（または最もシンプルなテンプレ）を追加する。モデルに `deepseek-r1:1.5b` を指定して、1 往復会話して応答が返れば最初の設定はクリア。

1.5 トラブルシューティング

`localhost:11434` を入れてしまう

Docker 内の localhost なので NG（Dify→Ollama へ届かない）

`curl http://<LAN_IP>:11434/api/tags` が返らない

- Ollama が 127.0.0.1 にしかバインドしてない
- `OLLAMA_HOST=0.0.0.0:11434 ollama serve` を実行し直す

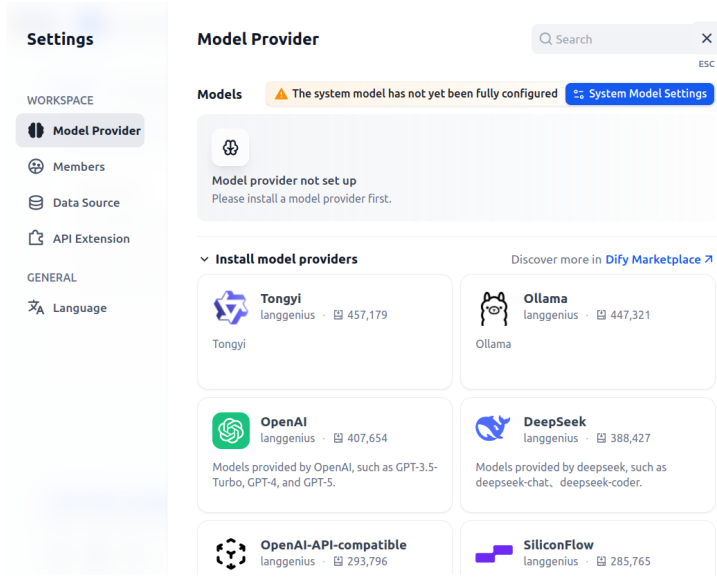

図 1.6: Dify モデル選択画面

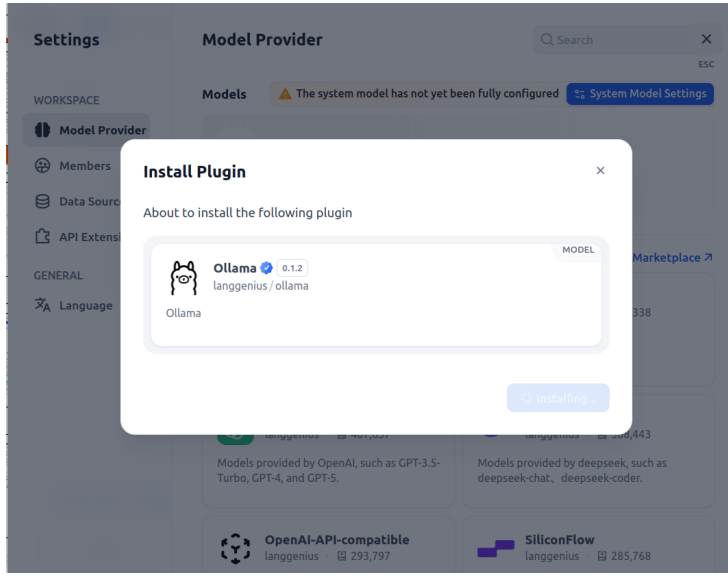

図 1.7: Dify モデル選択画面 - Ollama のインストール

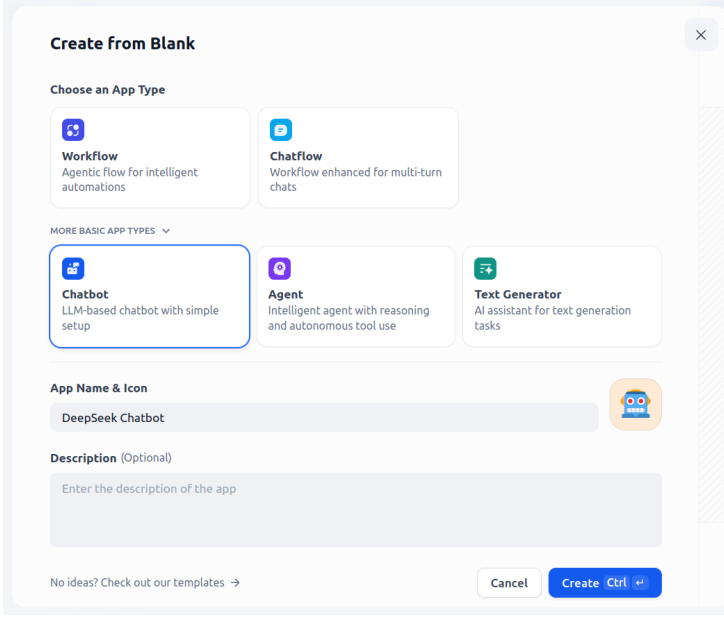

図 1.8: Dify アプリ作成 - 最小チャットボットの作成

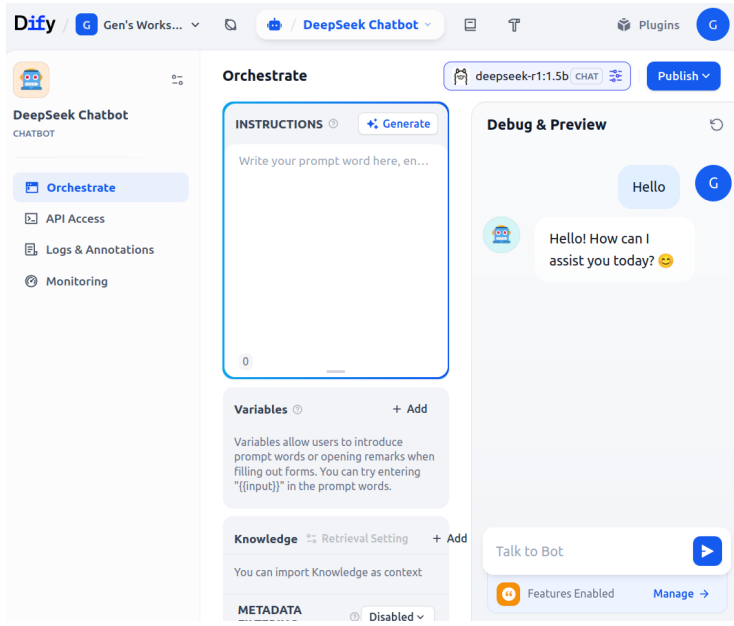

図 1.9: Dify アプリ作成 - 最小チャットボットの設定

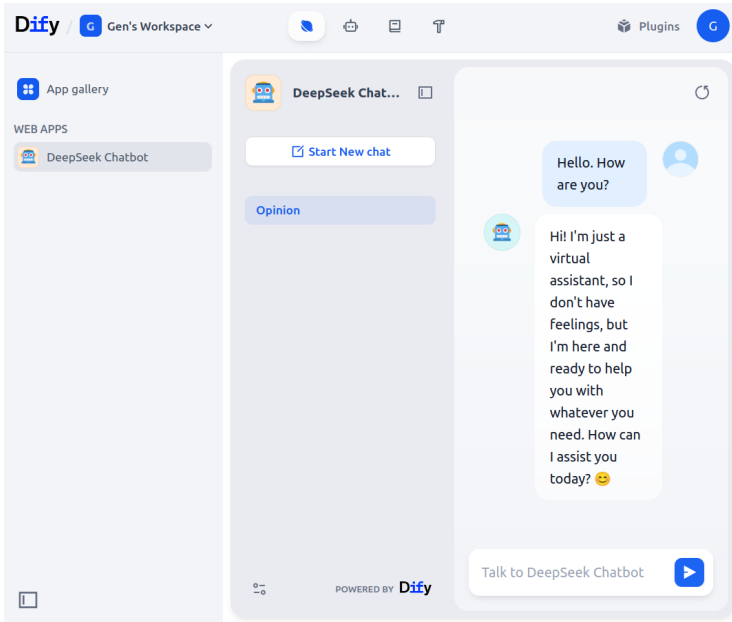

図 1.10: Dify アプリ作成 - 最小チャットボット

Dify から Ollama へ LAN_IP 経由で繋がらない

Dify コンテナ → Ollama(LAN_IP:11434) に TCP 接続できていない（タイムアウト）

- ファイヤーウォール

```
$ sudo ufw allow 11434/tcp
```

- Dify 上の Docker コンテナから確認

モデルが返れば成功

```
$ docker exec -it docker-api-1 sh -lc "apk add --no-cache curl >/
dev/null 2>&1 || true; curl -sS http://192.168.81.215:11434/api/
tags | head"
```

Dify インターフェース上から Ollama のモデルが見つからない。

モデルが見えるか確認する。スペルミス等ありえる。モデルの設定は.ollamaから読まれるが、実行ユーザー ollama とモデルの場所が違う場合にモデルが空になる。

- モデルが空の場合

```
{"models":[]}
```

- .ollamaパスをちゃんとリンクする。

```
$ ln -vs /usr/share/ollama/.ollama ~
```

- 確認

```
$ LAN_IP=$(ip route get 1.1.1.1 | awk '{print $7; exit}')
$ curl -s http://$LAN_IP:11434/api/tags

{"models":[{"name":"deepseek-r1:1.5b","model":"deepseek-r1:1.5b",
"modified_at":"2026-02-13T06:56:58.294031073+01:00","size"
:1117322768,"digest":"
e0979632db5a88d1a53884cb2a941772d10ff5d055aabaa6801c4e36f3a6c2d7"
,"details":{"parent_model":"","format":"gguf","family":"qwen2","
families":["qwen2"],"parameter_size":"1.8B","quantization_level":
"Q4_K_M"}}]}
```

1.6 参考リンク

1. Dify Docker Compose クイックスタート（公式）(docs.dify.ai Quick)
2. Dify × Ollama 統合（公式：Base URL の注意が明確）(legacy-docs.dify.ai Ollama)
3. Ollama FAQ（OLLAMA_HOST で公開できる）(Ollama Docs FAQ)
4. Ollama + DeepSeek + Dify のプライベートデプロイ (legacy-docs.dify.ai DeepSeek)

第 2 章

RAGの実装と検証

この章では PDF 文書 1 本を使用して **RAG が動く**ことを目標にします。何はともあれ動いて検証するということに重点をおいてパラメータを変えて挙動を見てみましょう。

RAG についての概略を説明し、Chunk_size、Top_k、Temperature、Embedding を触って差を観察し、どう精度が変わるのかを説明していきます。

2.1 RAG(Retrieval Augmented Generation)の概略

検索の流れ：質問 → Embedding（意味ベクトル化）→ ベクトル DB で類似検索 → 関連チャンク Top_k 件取得 → LLM へ渡す → 回答生成

RAG（Retrieval Augmented Generation）とは、大規模言語モデル（LLM）に外部知識を動的に参照させることで、回答の正確性と文脈適合性を高める手法である [1][2][3][4]。通常の LLM は事前学習されたパラメータの範囲内でのみ回答を生成するが、RAG では質問に応じて関連文書を検索し、その情報をコンテキストとして与えた上で生成を行う。これにより、モデル自体を再学習することなく、最新情報や社内限定情報を活用できる。

RAG の基本構造は大きく三段階に分かれる。

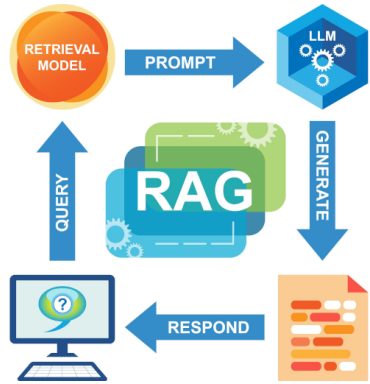

図 2.1: RAG 検索

1. 文書を適切な単位（Chunk）に分割し、それぞれを Embedding（意味ベクトル化）する。
2. ユーザーの質問も同様に Embedding し、ベクトル空間上で類似度検索を行う。
3. 上位の関連チャンク（Top_k）を LLM へ渡し、それらを参照しながら回答を生成する。

この流れは「検索（Retrieval）」と「生成（Generation）」の明確な責任分離によって成立している。

実務導入において RAG が重要視される理由は、モデルの再学習を行わずに企業独自の知識を活用できる点にある。社内マニュアル、契約書、技術仕様書などをナレッジとして登録することで、閉じた環境内で AI を活用できる。特にローカル LLM と組み合わせることで、データを外部に送信せずに業務支援 AI を構築することも可能である。

重要なのは、RAG は単なる「検索＋生成」ではないという点である。検索精度は Embedding モデルと Chunk 設計に依存し、生成精度はプロンプト設計および Temperature などの生成パラメータに依存する。つまり RAG の品質は、(1) 文書分割（Chunk）の妥当性、(2) 意味検索の精度（Embedding）、(3) 生成時の制御（Top_k）、という三つの設計要素のバランスで決まる。どれか一つが適合してないと、回答は不安定になる。**RAG は「導入すれば自動で賢くなる」技術ではなく、実務上では設計と検証を繰り返す試行錯誤による工学的対象になる。**

本書では、Dify を用いてローカル LLM 環境下で RAG を構築し、

Chunk 設計、Embedding 選択、Top_k 調整、Temperature 制御といった各要素がどのように精度へ影響するかを実験的に検証する。目的は単に動かすことではなく、動かして、調整し、なぜ精度が変わるのかを理解することにある。

用語説明：

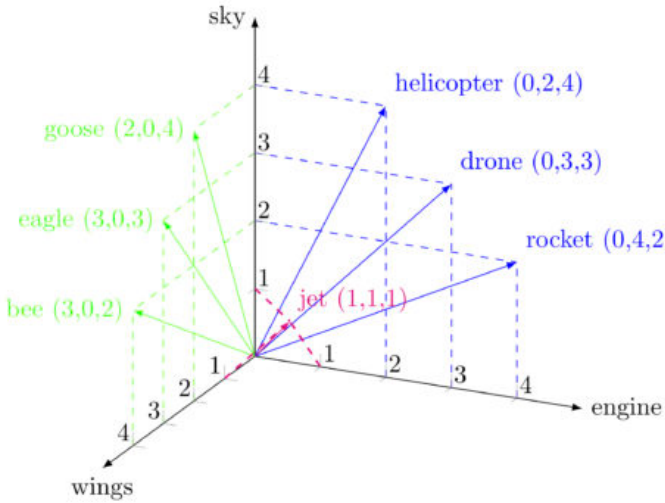

図 2.2: Embedding - テキストデータのベクトル化 (@corpling.hypotheses.org)

- Chunk：文脈の単位。元のドキュメントを**検索可能な単位に分割したテキスト片**のこと。RAG では PDF や文書全体をそのまま扱うのではなく、数百〜千文字程度の塊に分割して Embedding し、ベクトル検索する。チャンクが大きすぎるとノイズが混ざりやすく、小さすぎると文脈が壊れる。Chunk 設計は「文脈の保存」と「検索精度」のバランス設計のこと。RAG 精度は Chunk に大きく依存する。

- Top_k：情報量の制御。ベクトル検索で**上位何件のチャンクを LLM に渡すかを決めるパラメータ**のこと。例えば、$Top_k = 3$ なら、質問に最も近い 3 つのチャンクを取得する。小さすぎると重要情報を取りこぼし、大きすぎると無関係な情報が混入し、生成精度が落ちる。Top_k は「情報の網羅性」と「ノイズ制御」のトレードオフ調整レバーになる。

- Embedding：意味の変換。**テキストを意味を保持した数値ベクトルに変換する処理**のこと。RAG ではこのベクトル同士の距離（類似度）で関連文書を検索する。Embedding モデルが弱いと、意味検索が正しく働かず、質問と無関係なチャンクが選ばれる。特に日本語を扱う場合は、multilingual 対応モデルを選ぶことが精度に直結する。Embedding は RAG の “検索の質” を決める基礎になる。

- Temperature：表現の揺らぎ。LLM の**出力のランダム性（＝創造性）を制御する値**。低い（例: 0.1〜0.3）ほど決定的で安定した出力になり、高い（例: 0.7〜1.0）ほど多様で創造的になる。例えば、画像生成 AI では影響が顕著に見える（低いと同じような画像ばかりになり、高いとクリエイティブに感じる）。RAG 用途では通常、事実性が重要なため低めに設定するのが基本。Temperature は検索精度には影響せず、生成の表現と揺らぎに影響する。

2.2 Dify 上で Knowledge Base（知識基盤）を作る

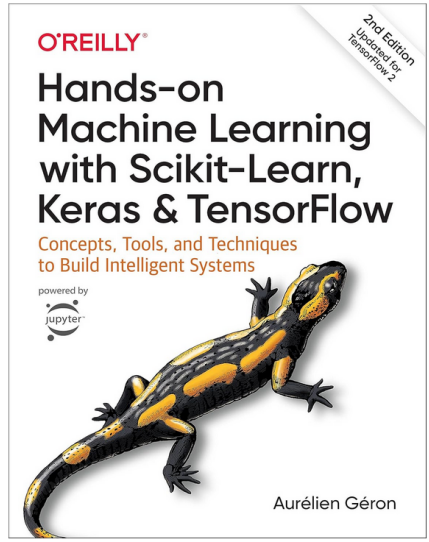

図 2.3: 知識基盤: Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

Dify ⇒ Knowledge ⇒ Create Knowledge ⇒ Import from file ⇒ PDF をアップロード

準備

- PDF を用意する（20〜30 ページ）。おすすめは「章がはっきりしていて用語が繰り返し出る」技術文書。（例：プロトコル解説、API 仕様の一部、社内手順書の抜粋など）
  - この例では、英文の O'Reilly の「Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow」の第１章（約３０ページ）を使用します。
- Knowledge Base 作成時に選ぶ “Chunk Mode” は後から変更できない。ただし区切り文字や最大長などのチャンク設定は調整可能。(docs.dify.ai - Chunk)
- TopK、Score Threshold、Rerank は「検索で拾うチャンクの量と質」を直接変える。(docs.dify.ai - Index Method)

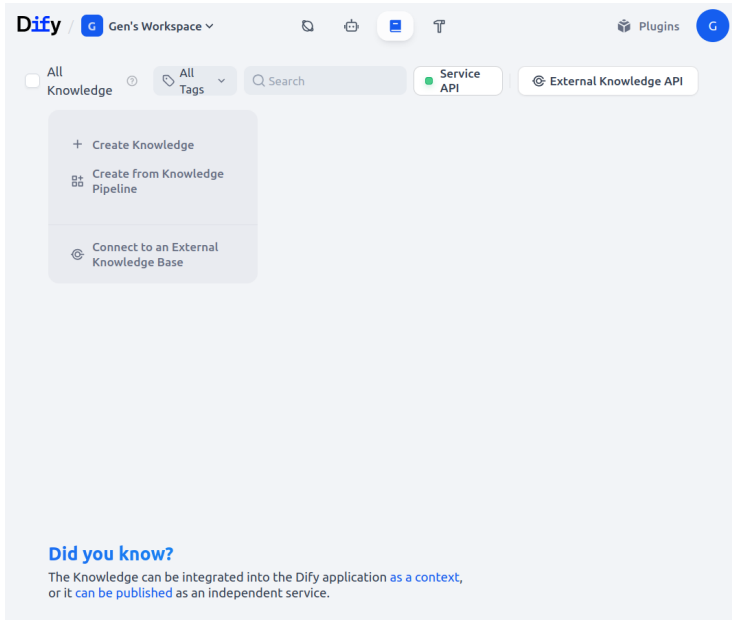

図 2.4: Create Knowledge を選択

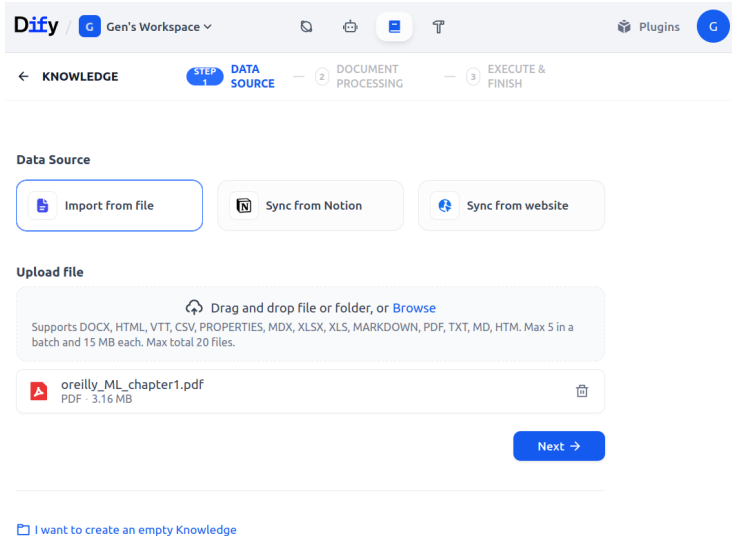

図 2.5: この例では Import File で PDF ファイルを選択

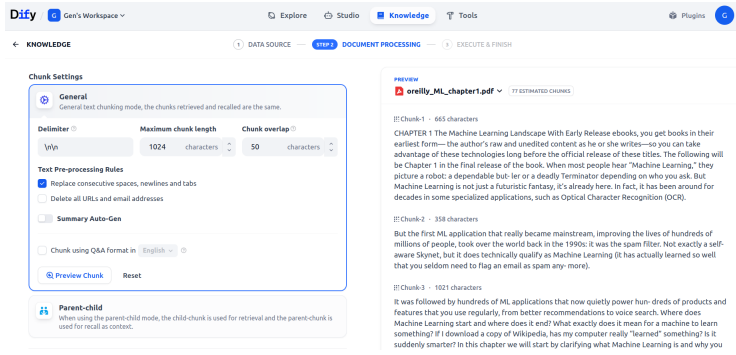

図 2.6: Chunk はデフォルトで 1024 を選択

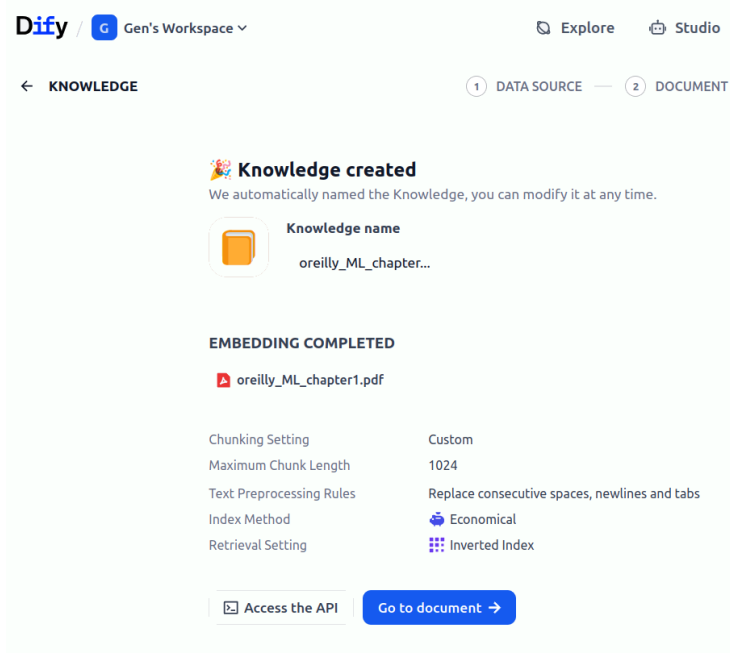

図 2.7: Knowledge が作成された

2.3 設定

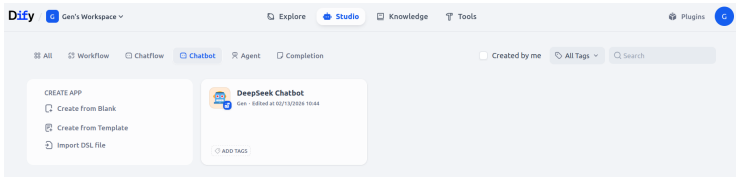

図 2.8: Chatbot から Knowledge を使う（Studit -> DeepSeek Chatbot を選択）

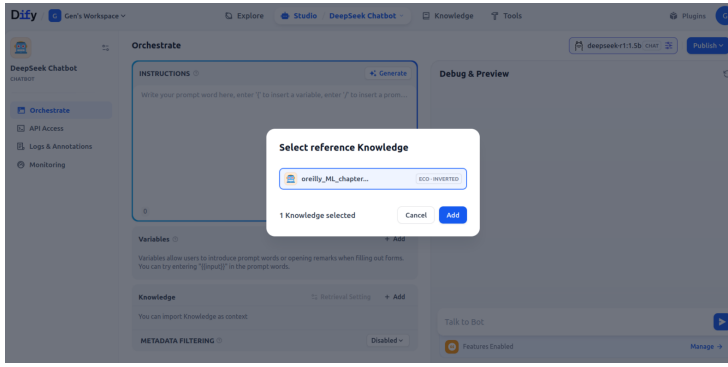

図 2.9: Chatbot から Knowlege を選択

Chunking 設定

「Chunking and cleaning text」画面で設定します。まずは以下の２パターンを同じ PDF ファイルで比較する。

1. 粗い：max chunk length 800〜1200 くらい
2. 細かい：max chunk length 300〜500 くらい

Dify は「各ドキュメントごとにチャンク設定を持てる」ので、同じ Knowledge 内でも比較しやすい。

Embedding モデルを選ぶ（最初は外部推奨）

日本語が混ざる可能性があるなら **multilingual 系**が無難。Dify チュートリアルでも言及がある。日本語対応の設計としては

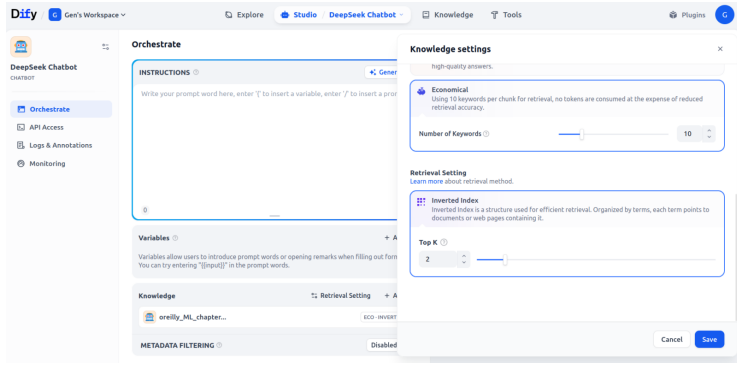

図 2.10: Chatbot の Top_K パラメタを設定（初期値は 2）

- 生成：Ollama deepseek-r1:1.5b（ローカル）
- Embedding（検索）：外部（multilingual 対応。例えば ChatGPT API）

等で行う。この例では O'Reilly の英文テキストを使うのでこの部分は深く考えない。Ollama から使える無料版の`bge-m3`を使用する。

設定方法

1. モデルをインストール

```
$ ollama pull bge-m3
```

2. Dify 上で "Settings" → "Model Provider" → "Ollama" → "Add Model" で選択。設定項目は "Embedding" と "bge-m3" を選択する以外は DeepSeek Chatbot とほぼ同様。
3. Embedding モデルを変更したら "Knowledge" からもう一度 "Save & Process" ボタンを押して、念の為 Index を再度作成しておく。

Retrieval 設定（TopK / Score Threshold / Rerank）

Knowledge 作成後（または設定画面）で **Indexing method / Retrieval settings** を触る。

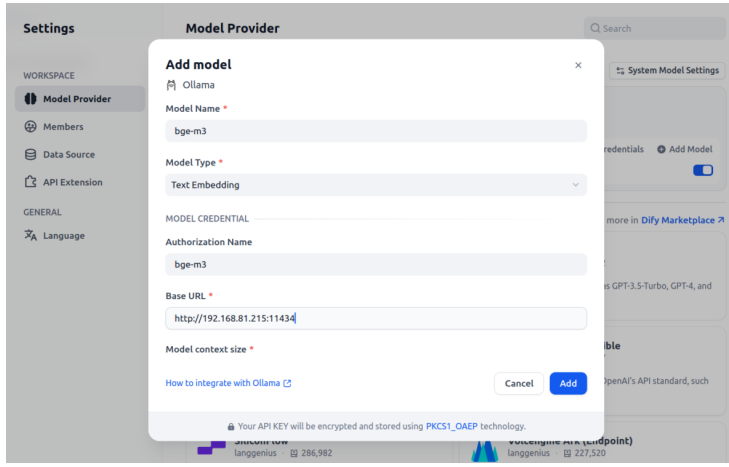

図 2.11: Embedding Model（bge-m3）を Ollama 上から選択

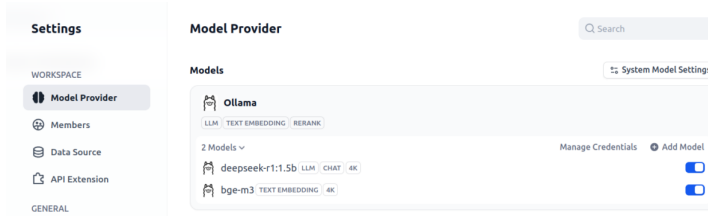

図 2.12: Embedding Model の追加完了

TopK の比較（最低 2 点）

- **TopK = 2**
- **TopK = 6**

TopK は「拾うチャンク数」。小さいと取りこぼし、大きいとノイズ混入が増えやすい、というトレードオフ。

Score Threshold

0.3 / 0.5 / 0.7 あたりで試す。これは、高いほど厳選、低いほど拾う。

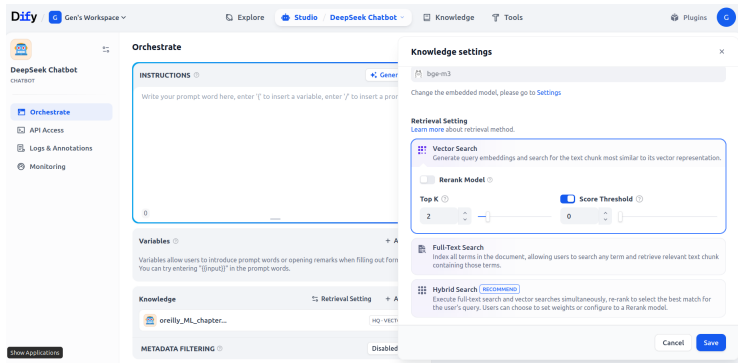

図 2.13: “DeepSeek Chatbot” の “Knowledge settings” で Score Threshold など変更可能。この場合は High Quality Index Method = bge-m3 を選択。

Rerank

Rerank が使える構成なら「TopK で多めに拾って、Rerank で上位を整列」が効きます（使えるモデルがあれば）。Dify のモデル種別として rerank が存在する点は公式にも説明があります。

2.4 検索テスト

Dify の「Retrieval Test」で “検索の質” を評価

非常に重要な項目といて、Knowledge（データセット）には **テスト検索（Retrieve / Test retrieval）** があり、まずここで「検索が正しく当たってるか」を見ます。

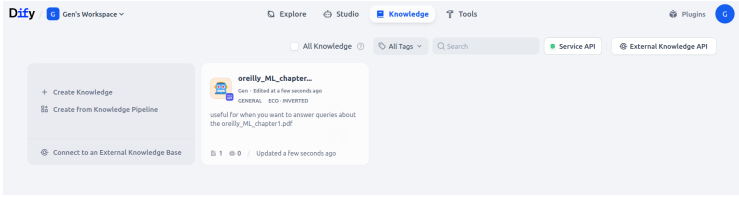

図 2.14: Knowledge を確認

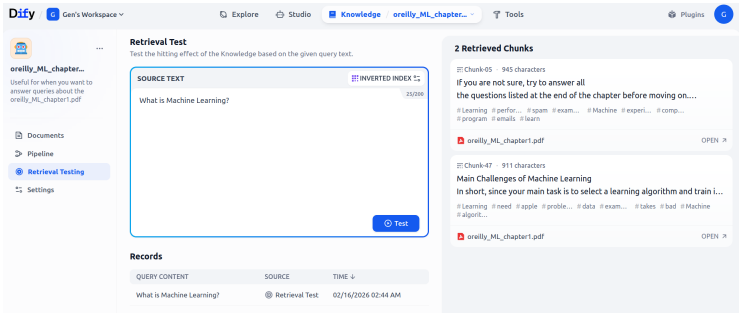

図 2.15: Knowledge から Retrieval Testing（検索テスト）で妥当な文脈が選択されているかチェック

テスト用クエリ例（PDF が技術文書の場合）

文書として機械学習の本の第一章を与えているので、その知識を問い合わせてみます。

1. “What is Machine Learning according to Hands-on Machine Learning book?”
2. “What is the difference between supervised learning and unsupervised learning?”
3. “What is the difference between batch learning and online learning?”
4. “What is overfitting and how can it be reduced?”
5. “What is the purpose of a validation set in Machine Learning?”

ここで見るべき観点

- 上位チャンクが「質問と同じ節」から取れているか
- 余計な章が混ざっていないか
- チャンクが短すぎて文脈が欠けていないか（細かすぎ問題）
- チャンクが長すぎて要点が埋もれていないか（粗すぎ問題）

2.5 使ってみる

アプリ側（チャットボット）で temperature を比較する

前章で作った DeepSeek チャットアプリ（Ollama）に Knowledge を紐づけて（“Integrate knowledge within apps”）RAG 回答させます。

temperature 比較はこの 2 点で行う

1. **temp = 0.2**（堅め：根拠重視）
2. **temp = 0.7**（柔らかめ：言い回し増える）

RAG で重要なのは「検索が当たってる」ことなので、temperature 設定は最後に触る。

任意：API で Retrieval を叩いて再現性を上げる

Dify には **Dataset retrieve（テスト検索）API** がある。UI で見た結果を、API でも同条件で再現できる。

2.6 検証結果のまとめ

```
# RAG検証ログ

## PDF
- タイトル：Hands-on Machine Learning with Scikit-Learn, Keras &
TensorFlow, O'Reilly
- ページ数：33
- 想定用途：FAQ / 手順 / 仕様参照

## 実験パラメータ
### Chunk
- A: max chunk length = 1024
- B: max chunk length = 512

### Retrieval
- TopK: 2 / 6
- Score threshold: 0.3 / 0.5 / 0.7
- Rerank: off

### Embedding
- provider/model: "bge-m3"

### Generation
- LLM: deepseek-r1:1.5b (Ollama)
- temperature: 0.2 / 0.7

## テストクエリ（5個）
1. "What is Machine Learning according to Hands-on Machine
Learning book?"
2. "What is the difference between supervised learning and
unsupervised learning?"
3. "What is the difference between batch learning and online
learning?"
4. "What is overfitting and how can it be reduced?"
5. "What is the purpose of a validation set in Machine Learning?"

## 観察結果（結論から）
- 一番良かった組み合わせ：Chunk 1024, Top_K 6, Threashold 0.3,
Temperature 0.2
- 悪かった組み合わせ：Chunk 1024, Top_K 2, Threashold 0.7,
Temperature 0.7

Top_Kである程度広い範囲を検索結果に出してThreasholdで厳選する方が原
文に忠実かつ適度に要約してくれる。Citationも出す。Temperatureも低め
```

の方が無用な創造性を発揮せず手堅い。

逆に、Top_Kが低く、Threashold高めでTemperature（創造性大）であると無駄な単純化や構成が多くなる。現状ではまだ好みの問題である。ただし、大量の文書を入力した場合に、正しくKPIを設定してないとユーザーによっては不満が募るはず。

なぜ精度が変わるのか?
- チャンクが粗い → （例：ノイズ増/要点が埋まる）
- チャンクが細かい → （例：文脈欠落/断片化）
- TopKが小さい → 取りこぼし
- TopKが大きい → 低関連チャンク混入
- threshold高い → 厳選されるが漏れる
- threshold低い → 拾うがノイズ増
- embeddingが合わない → 意味検索が弱い（特に多言語）

2.7 参考リンク

1. Chunking and cleaning text（チャンク設定）(docs.dify.ai - Chunk)
2. Setting indexing methods（TopK/threshold など）(docs.dify.ai - Index Method)
3. Integrate knowledge within apps (docs.dify.ai - Within Apps)
4. Retrieval test & TopK の意味（補助）(raglegacy-docs.dify.ai - Retreval Test)
5. Retrieve chunks API（任意で再現性）(ragdocs.dify.ai - Retrieve Chunks)

第 3 章

RAG の理論構造と実践理解

- Embedding の数学的意味
- ベクトル空間とは何か
- 類似度計算
- 検索と生成の責任分離

3.1 RAG とは何か?

- Retrieval + Generation の分離
- Vector DB の役割
- なぜ Embedding が必要か
- 生成と検索の責任分離

3.2 チャンク設計の思想

- chunk_size の意味
- 文脈破壊のリスク
- 日本語文書特有の課題

3.3 TopK とノイズ問題

- 小さすぎる問題
- 大きすぎる問題
- Rerank の意味

3.4 精度検証の方法論

- Retrieval test
- 誤答パターン分類
- 再現性のある検証

第 4 章

Dify 内部構造の理解

4.1 Dify のアーキテクチャ

- API 層
- Worker
- Redis
- Postgres
- Weaviate

4.2 モデルプロバイダの仕組み

- OpenAI 型
- Ollama 型
- DeepSeek API 型

4.3 Plugin / Tool 呼び出しの内部

4.4 なぜ Dify は LangChain と違うのか

第 5 章

実務導入パターン

5.1 社内 FAQ 型

- ナレッジ限定
- 低リスク

5.2 技術文書検索型

- 精度要求高い
- Chunk 設計重要

5.3 契約書・法務型

- セキュリティ重視
- オンプレ優先

5.4 エージェント型

- ワークフロー
- 外部 API 連携

5.5 AI タイプ別整理

- 単純 QA 型
- ワークフロー型
- エージェント型
- 推論特化型（DeepSeek 等）

第 6 章

開発スタイル

6.1 なぜウォーターフォールは失敗するか

6.2 PoC（Proof-Of-Concept）主導開発

- 概念実証モデルの重要性
- 小さく動かす
- 期待値調整

6.3 RAG のアジャイル改善

- データ改善
- Prompt 改善
- Retrieval 調整

6.4 Scrum との統合

- スプリント単位の評価指標
- Definition of Done

6.5 精度 KPI の設定方法

第 7 章

セキュリティとガバナンス

7.1　外部 API のリスク

7.2　ローカル LLM の限界

7.3　ハイブリッド構成

7.4　データ分類と AI 利用ポリシー

7.5　監査・ログ設計

第 8 章

将来展望

8.1　何ができるのか？

8.2　マルチエージェント化

8.3　自律型 AI

8.4　日本市場の可能性

8.5　小規模 LLM の未来

第 9 章

TODO

9.1 資料作成補助（スライド等）AI

9.2 画像生成 AI

9.3 音声生成 AI

9.4 動画生成 AI

9.5 面接・人材採用システム用ワークフロー

第 10 章

Appendix: ＤｉｆｙをＡＷＳで使う

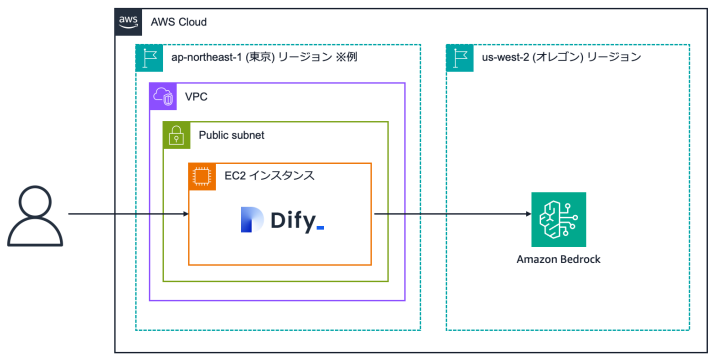

図 10.1: AWS 上の Dify の構成図（出典: dify.aws.studio）

注意点

Dify を AWS 上にデプロイする試験環境は、自身の AWS アカウントでも実施可能です。ただし、実運用を見据える場合には権限設計・リージョン選択・モデル選択を慎重に考える必要がある。

10.1 AWS アカウントで実施する場合の前提条件

図 10.2: AWS へのログイン）

試験的に色々試す時には、コンソールログイン時の IAM ユーザーまたはロールに`AdministratorAccess`の権限が必要です。実際の本番環境では **最小権限の原則（Principle of Least Privilege）**を適用することが AWS のベストプラクティスになる。

同一アカウントで複数人が同時に Dify 環境を立ち上げると、CloudFormation リソース競合、VPC や IAM ロールの重複エラー、Bedrock クォータ超過などが発生する可能性があります。対応策としては、

- 代表者 1 名のみデプロイ
- Dify 内で複数ユーザーを払い出す
- CloudFormation テンプレートから重複リソースを除外
- 環境ごとにスタック名を変更する

Enterprise 環境では、**アカウント分離（開発・検証・本番）**が推奨されます。

10.2 リージョン選択

Dify を構築するリージョン

任意のリージョンで可能だが、このテスト構築では **ap-northeast-1（東京）**、または **us-west-2（オレゴン）**を利用する。オレゴンの利点は、利用可能モデルが多く、クォータが高く、最新モデルが早く提供されるということにある。最新モデルを試したい場合には最適である。

ただし、企業利用の場合は、

- データ主権（Data Residency）
- GDPR / 個人情報保護
- 企業セキュリティポリシー

を考慮して選択する必要がある。仮に欧州の企業であれば、GDPR の制約から通常は欧州リージョン内で完結することが要望されることが多い。日本の企業であっても日本内で完結しておくことがセキュリティなどの観点から望ましい。

例えば、日本の企業であっても、以下が制約として課されることがある。

- 海外リージョン禁止
- クロスリージョン推論禁止
- データ国外転送制限

そのため、**モデル戦略は組織ポリシーに依存**することになる。

ユースケース別モデル選択戦略

以下は、制約別の推奨モデル構成です。

日本リージョンのみ（クロスリージョン推論可能）

用途	推奨モデル
高度テキスト処理	Claude 3.5 Sonnet v2（クロスリージョン推論）
軽量テキスト処理	Claude 3 Haiku
Embedding	Titan Text Embeddings V2
Rerank	Cohere Rerank 3.5

日本リージョンのみ（クロスリージョン不可）

用途	推奨モデル
高度テキスト処理	Claude 3.5 Sonnet
軽量テキスト処理	Claude 3 Haiku
Embedding	Titan Text Embeddings V2
Rerank	Cohere Rerank 3.5

この場合、モデル性能は若干制約を受ける。

海外リージョン利用可能（推奨構成：オレゴン）

用途	推奨モデル
Chat/Agent/高度テキスト処理	Anthropic Claude 3.5 Sonnet v2
軽量テキスト処理	Anthropic Claude 3.5 Haiku
Embedding	Amazon Titan Text Embeddings V2
Rerank	Cohere Rerank 3.5

この構成が最も高性能になる。

10.3 Enterprise 視点での重要ポイント

IAM 設計を必ず再構築する

試験用`AdministratorAccess`を**そのまま本番には使わないこと**。

- サービスロール分離
- Bedrock 専用ポリシー
- VPC エンドポイント制限
- CloudWatch ログ監査

を設計する必要があります。

モデル選択はセキュリティポリシー依存

企業では以下を確認する。特に法務関係等では制約が格段に上がる。

- データは海外リージョンへ送信可能か
- 推論ログは保存されるか

- Bedrock 利用規約との整合性
- PII 処理可否

クォータ管理

Bedrock はリージョンごとにクォータがあります。

- トークン制限
- 同時実行数
- モデル別制限

試行段階で必ず確認しましょう。

10.4 参考リンク

1. Dify での生成 AI アプリケーション構築ワークショップ（AWS Workshop Studio）(dify.aws.studio)

第 11 章

Appendix: Dify Docker を Ubuntu で使う

多少古いディストリビューションですが、Ubuntu20 は安定しているのでこの版で Docker Compose V2 が動作するようにします。クラウド環境だとそのまま最新の Docker やモジュールをインストールすれば良いでしょう。

Docker の事前環境

```
docker --version
docker-compose version
```

出力例

```
$ docker --version
Docker version 24.0.5, build 24.0.5-0ubuntu1~20.04.1

$ docker-compose version
docker-compose version 1.25.0, build unknown
docker-py version: 4.1.0
CPython version: 3.8.10
OpenSSL version: OpenSSL 1.1.1f 31 Mar 2020
```

古い Docker 関連を整理（安全）

```
sudo apt-get remove docker docker-engine docker.io containerd
runc
```

（データは消えない）

Ubuntu20 の標準へ戻す場合

```
sudo apt-get install docker.io
```

公式 Docker 版

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker
-compose-plugin
```

完全削除

```
sudo apt-get purge docker.io
sudo rm -rf /var/lib/docker
```

これをやらない限りデータは残る。apt-get remove ではデータは残るのが普通。

公式 Docker リポジトリ追加

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl gnupg

sudo install -m 0755 -d /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/
keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(
lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/
docker.list > /dev/null
```

最新 Docker Compose v2 をインストール

```
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

確認

```
docker --version
docker compose version
```

出力

```
Docker version 28.1.1, build 4eba377
Docker Compose version v2.35.1
```

これで Dify Docker を動かす準備が整いました。

参考文献

[1] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *arXiv preprint arXiv:2005.11401*, 2020, Available: https://arxiv.org/abs/2005.11401

[2] Facebook AI Research, "Retrieval-augmented generation: Streamlining the creation of intelligent natural language processing models." 2020.

[3] Dify Documentation, "Knowledge & rag introduction." https://docs.dify.ai/en/use-dify/knowledge/readme#introduction, 2026.

[4] OpenAI, "Embeddings guide." https://platform.openai.com/docs/guides/embeddings, 2026.

あとがき