

Documentation Technique du Chatbot RAG

Table des matières

1. [Vue d'ensemble](#)
2. [Architecture du système](#)
3. [Modules et composants](#)
 - [app.py](#)
 - [rag_chain.py](#)
 - [retriever.py](#)
 - [api_client.py](#)
 - [utils.py](#)
 - [templates/index.html](#)
 - [static/css/style.css](#)
 - [static/js/main.js](#)
4. [Scripts de collecte de données](#)
 - [get_posts.py](#)
 - [guides.py](#)
5. [Flux de données](#)
6. [Modèles et embeddings](#)
7. [Interface utilisateur](#)
8. [Configuration et déploiement](#)
9. [Points d'extension](#)

Vue d'ensemble

Pour mon travail de bachelor, j'ai mis au point un chatbot de support technique basé sur l'architecture Retrieval-Augmented Generation (RAG). Pour le contenu, j'ai associé des guides techniques détaillés à des discussions issues de Reddit, afin de proposer des réponses à la fois précises et contextuelles.

Sur le plan technique, j'ai utilisé LangChain pour piloter le pipeline et j'ai converti les documents en vecteurs via les embeddings Hugging Face, avant que FAISS n'indexe ces représentations pour extraire les passages pertinents. Enfin, GPT-4.1 génère la réponse finale, et l'application, déployée sous Flask, expose une API REST et une interface web en HTML/CSS.

Architecture du système

J'ai préféré découper le chatbot en six blocs bien séparés. Cette approche me permet de localiser rapidement un bug ou de faire évoluer une partie du code sans démonter tout le reste.

1. Interface web (**app.py** + **templates/static**)

- Dans **app.py**, j'initialise Flask, je déclare les routes et je traite les requêtes HTTP.
- Les dossiers **templates** et **static** concentrent tout ce qui est visuel : pages HTML, feuilles de style et scripts.
- En bref, c'est la porte d'entrée du chatbot, tant pour l'utilisateur que pour les appels REST.

2. Moteur RAG (`rag_chain.py`)

- Dès qu'une question arrive, je la reformule en plusieurs variantes pour multiplier les angles de recherche.
- Ces formulations partent ensuite vers le retriever ; les passages retenus sont transmis à GPT-4.1 pour générer la réponse.
- On peut voir ce fichier comme le chef d'orchestre de tout le pipeline.

3. Système de récupération (`retriever.py`)

- Chaque document est d'abord transformé en vecteur grâce aux embeddings Hugging Face.
- FAISS gère ensuite l'indexation et renvoie les segments les plus proches sur le plan sémantique.
- Ce module fait le lien entre la base de connaissances et la question posée.

4. Client API (`api_client.py`)

- Pour compléter le corpus interne, je récupère des données auprès de services externes (notamment iFixit pour récupérer les étapes d'un guide).
- Les réponses JSON sont nettoyées, mises au bon format puis ajoutées à l'index.
- L'idée est d'éviter une base figée et d'intégrer des contenus régulièrement mis à jour.

5. Utilitaires (`utils.py`)

- Je centralise ici tout ce qui sert un peu partout : nettoyage de texte, conversions, logs, etc.
- C'est un gain de temps et - surtout - ça m'évite de répéter le même code.

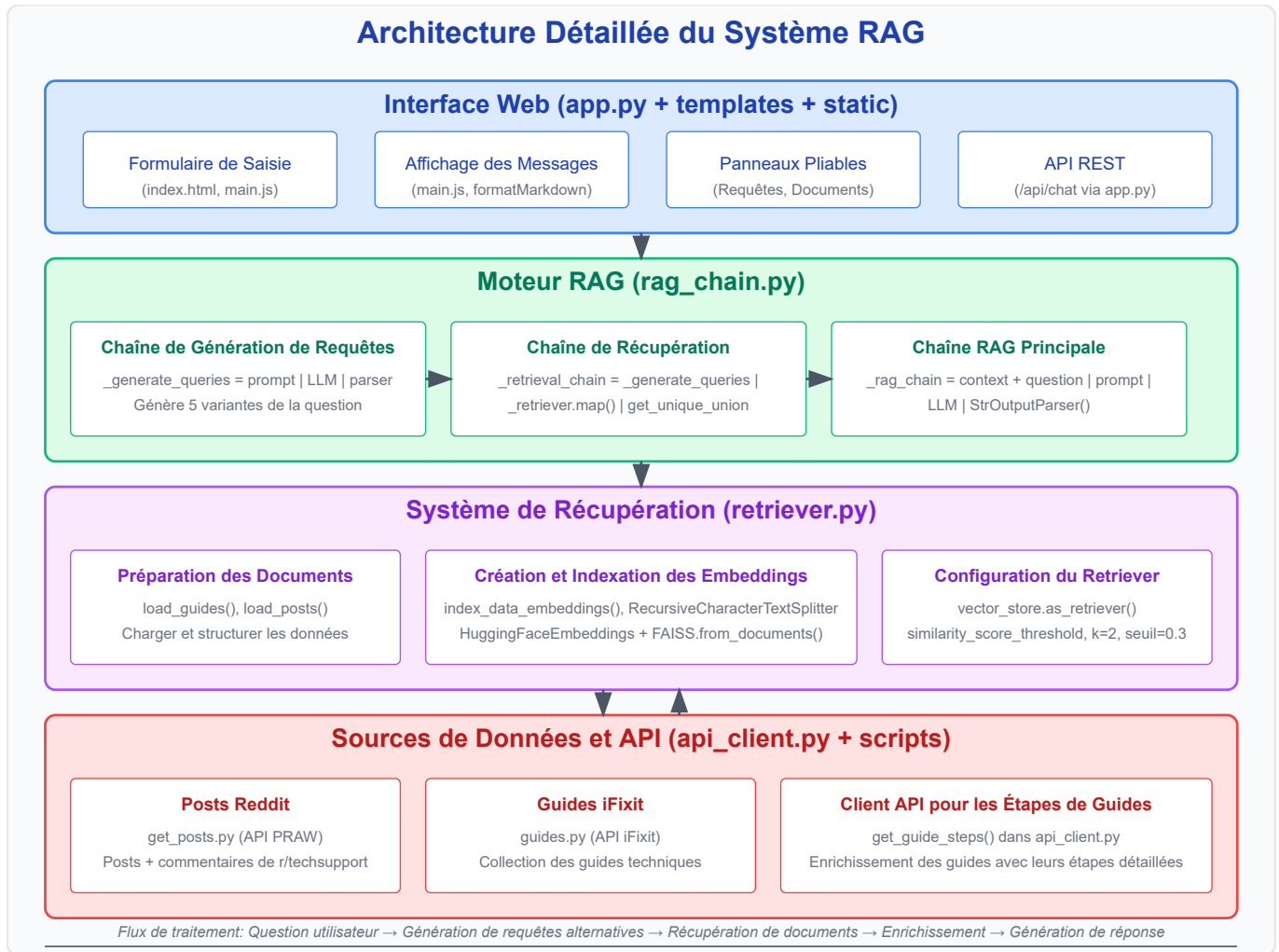
6. Interface frontend (`templates/index.html` + `static/css/style.css`)

- `index.html` contient un champ de saisie, un bouton « Envoyer » et une zone de réponse.
- `style.css` assure une mise en page sobre : couleurs neutres, typographie lisible, espacements aérés.

Comment tout s'enchaîne ?

1. L'utilisateur pose sa question.
2. Le moteur RAG crée des variantes et lance la recherche.
3. Le retriever interroge FAISS et récupère les extraits pertinents.
4. Ces extraits sont passés à GPT-4.1, qui rédige la réponse.
5. La réponse revient par l'API REST et s'affiche sans délai dans l'interface.

Cette séparation des rôles me permet de tester ou de remplacer un module (changer d'indexeur, par exemple) sans perturber l'ensemble du système.



Modules et composants

app.py

Ce module principal initialise l'application Flask et expose les endpoints REST pour interagir avec le chatbot.

Points d'entrée API:

GET /

```
@app.route("/")
def index():
    return render_template("index.html")
```

Sert la page d'accueil du chatbot.

POST /api/chat

```
@app.route("/api/chat", methods=["POST"])
def chat():
    start_time = time.time()
```

```

data = request.json
question = data.get("message", "")

if not question:
    return jsonify({"error": "Message vide"}), 400

# Récupération des chaînes de traitement
rag_chain = get_rag_chain()
generate_queries = get_generate_queries()
retrieval_chain = get_retrieval_chain()

# Génération des requêtes alternatives
queries = generate_queries.invoke(question)

# Récupération des documents
retrieved_docs = retrieval_chain.invoke(question)
formatted_docs = format_documents(retrieved_docs)

# Génération de la réponse
response = rag_chain.invoke(question)

elapsed_time = time.time() - start_time

return jsonify({
    "response": response,
    "queries": queries,
    "documents": formatted_docs,
    "processing_time": f"{elapsed_time:.2f} secondes",
})

```

Endpoint principal

Ce point d'accès assure le traitement complet d'une requête utilisateur :

- Accepte la question envoyée en JSON ;
- Génère des requêtes alternatives pour améliorer la récupération ;
- Récupère les documents pertinents dans l'index FAISS ;
- Produit une réponse détaillée à l'aide de GPT-4.1 ;
- Renvoie à l'utilisateur la réponse, les requêtes générées, les documents utilisés et le temps de traitement global.

Initialisation:

```

if __name__ == "__main__":
    # Ne pas initialiser deux fois avec le reloader
    if os.environ.get("WERKZEUG_RUN_MAIN") == "true":
        with app.app_context():
            initialize_rag_system()

    # Vérifier si CUDA est disponible

```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
logger.info(f"Utilisation du périphérique: {device}")

app.run(debug=True, host="0.0.0.0", port=5000)
```

Initialise le système RAG et démarre le serveur Flask.

rag_chain.py

Ce module contient la logique principale du système RAG, configurant les chaînes de traitement LangChain pour orchestrer la récupération et la génération.

Fonctions principales:

initialize_rag_system()

```
def initialize_rag_system():
    """Initialise le système RAG avec les chaînes de traitement"""
    global _retriever, _rag_chain, _generate_queries, _retrieval_chain

    # Charger les variables d'environnement
    load_dotenv()
    OPENAI_KEY = os.getenv("OPENAI_KEY")

    # Charger les données
    posts = load_posts("./data/techsupport_posts.json")
    guides = load_guides("./data/guides.json")

    # Créer le retriever
    _retriever = index_data_embeddings(posts, guides)

    # Initialiser le LLM
    llm = ChatOpenAI(openai_api_key=OPENAI_KEY, model="gpt-4.1", temperature=0.0)

    # [Configuration des chaînes]

    # Chaîne de génération de requêtes
    _generate_queries = (
        prompt_template | llm | StrOutputParser() | (lambda x: x.split("\n"))
    )

    # Chaîne de récupération
    _retrieval_chain = _generate_queries | _retriever.map() | get_unique_union

    # Chaîne RAG complète
    _rag_chain = (
        {
            "context": _retrieval_chain | format_documents,
            "question": RunnablePassthrough(),
        }
        | final_prompt_template
```

```
        | llm  
        | StrOutputParser()  
    )
```

Avant de pouvoir répondre à la moindre question, j'exécute une fonction d'initialisation qui met en place l'ensemble du pipeline :

- Importe les sources brutes (guides techniques et posts Reddit);
- Paramètre le retriever en calculant les embeddings et en alimentant l'index FAISS;
- Établit les prompts de base destinés au LLM;
- Met en place la chaîne de reformulation chargée de créer des requêtes alternatives;
- Assemble la chaîne RAG principale, qui relie la recherche documentaire à la génération de réponses.

Prompts système:

Le système utilise deux prompts principaux:

1. Prompt de génération de requêtes alternatives:

Ta tâche est de générer cinq reformulations différentes de la question posée par l'utilisateur afin de retrouver des documents pertinents dans une base de données vectorielle.

En proposant plusieurs perspectives sur la question, ton objectif est d'aider l'utilisateur à surmonter certaines limites de la recherche par similarité basée sur la distance.

Fournis ces questions alternatives, chacune séparée par un saut de ligne.

Répond en Anglais

Question initiale : {question}

2. Prompt principal RAG:

L'utilisateur pose la question suivante :

➡ {question}

Tu disposes uniquement des documents suivants : des guides techniques et des posts Reddit pertinents.

Ces contenus incluent des descriptions générales, des conseils pratiques, des solutions proposées par la communauté, et parfois des instructions techniques détaillées.


🎯 Ta mission :

Base strictement ta réponse sur les informations présentes dans les documents

fournis ci-dessous ({context}).


N'utilise aucune connaissance extérieure. Si une information n'est pas présente, indique-le explicitement.

Fournis une réponse structurée, professionnelle et en français.

 Sources disponibles :

{context}

 Format de réponse attendu :

 Analyse du problème :


[Présente une synthèse du problème posé, uniquement en te basant sur les documents.]

☒ Vérifications préalables recommandées :

[Liste les éléments à inspecter ou tester avant toute manipulation, tels que suggérés dans les documents.]

 Procédure détaillée proposée :

[Structure la procédure étape par étape : "Étape 1", "Étape 2"... en t'appuyant sur les guides ou les conseils Reddit.]

 Conseils ou précautions à prendre :

[Ajoute ici uniquement les recommandations explicitement mentionnées dans les documents.]

 Sources consultées :

[Liste les URL des documents (guides ou posts Reddit) utilisés pour construire la réponse, selon les métadonnées disponibles.]

 Important :

Tu dois strictement t'appuyer sur les contenus fournis dans {context}. Aucune inférence ou ajout personnel n'est autorisé. Si la réponse n'est pas déductible des documents, indique-le clairement.

Configuration du LLM et contraintes d'information

- J'exécute GPT-4.1 avec une température réglée à 0,0, ce qui garantit des réponses déterministes et donc comparables d'un appel à l'autre.
- Le prompt final rappelle explicitement au modèle qu'il doit s'appuyer uniquement sur les passages remontés par le retriever ; toute connaissance externe est proscrite pour préserver la traçabilité des sources.

- Pour vérifier que cette règle est bien respectée, j'active la fonction `debug_context`, qui affiche dans les journaux le contexte envoyé au LLM ; je peux ainsi contrôler en un clin d'œil quels documents ont servi à la génération.

retriever.py

Ce module gère le chargement des données, leur préparation et l'indexation pour la recherche vectorielle.

Fonctions principales:

`load_guides()` et `load_posts()`

```
def load_guides(file_path):
    """Charge les guides depuis un fichier JSON"""
    try:
        with open(file_path, "r", encoding="utf-8") as f:
            guides = json.load(f)
            logger.info(f"Guides chargés avec succès: {len(guides)} guides trouvés")
            return guides
    except Exception as e:
        logger.error(f"Erreur lors du chargement des guides: {e}")
        return []

def load_posts(file_path):
    """Charge les posts depuis un fichier JSON"""
    try:
        with open(file_path, "r", encoding="utf-8") as f:
            posts = json.load(f)
            logger.info(f"Posts chargés avec succès: {len(posts)} posts trouvés")
            return posts
    except Exception as e:
        logger.error(f"Erreur lors du chargement des posts: {e}")
        return []
```

Ces fonctions chargent les données sources (guides et posts Reddit) depuis les fichiers JSON.

`index_data_embeddings()`

```
def index_data_embeddings(posts, guides, model_name="sentence-transformers/all-
MiniLM-L6-v2"):
    """Convertit les posts et guides en vecteurs et crée un retriever LangChain"""

    # Construire les objets Document pour les posts et guides
    documents = []
    # [Code de création des documents]

    # Découper les documents en chunks
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
    chunk_overlap=200)
```



```

splits = text_splitter.split_documents(documents)

# Créer des embeddings avec LangChain
embedding_model = HuggingFaceEmbeddings(
    model_name=model_name,
)
vector_store = FAISS.from_documents(splits, embedding_model)

# Retourner le retriever configuré
return vector_store.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k": 2, "score_threshold": 0.3},
)

```

Avant d'interroger la base, je dois d'abord transformer les sources brutes pour qu'elles soient digestes pour le système:

- Conversion en objets Document : je verse chaque guide ou message Reddit dans la classe Document de LangChain, ce qui homogénéise contenu et métadonnées.
- Découpage en morceaux : pour conserver une recherche fine, je scinde ensuite chaque texte en blocs d'environ **N** tokens; un passage trop long finirait par diluer l'information utile.
- Création des embeddings : ces blocs passent dans le modèle Hugging Face choisi afin d'obtenir leurs vecteurs d'embeddings, indispensables pour la comparaison sémantique.
- Indexation dans FAISS : j'insère ensuite les vecteurs dans un index FAISS, qui se chargera de retrouver les blocs les plus proches lorsqu'une question sera posée.
- Renvoi du retriever : pour terminer, je renvoie un objet retriever déjà connecté à cet index, prêt à être utilisé dans le reste du pipeline.

api_client.py

Ce module gère les interactions avec l'API externe iFixit pour récupérer les étapes détaillées des guides.

Fonction principale:

get_guide_steps()

```

def get_guide_steps(guideid):
    """Récupère les étapes d'un guide depuis l'API iFixit"""
    url = f"https://www.ifixit.com/api/2.0/guides/{guideid}"

    try:
        response = requests.get(url)
        if response.status_code != 200:
            logger.warning(f"Échec de récupération du guide {guideid}, code: {response.status_code}")
            return {"error": f"Échec de récupération du guide {guideid}, code: {response.status_code}"}

        data = response.json()

```

```

steps = []
cpt_steps = 0

for step in data.get("steps", []):
    cpt_steps += 1
    step_texts = [
        line["text_rendered"]
        for line in step.get("lines", [])
        if "text_rendered" in line
    ]
    steps.append({"stepno": cpt_steps, "text": step_texts})

logger.info(f"Guide {guideid} récupéré avec succès: {len(steps)} étapes")
return steps

except Exception as e:
    logger.error(f"Erreur lors de la récupération du guide {guideid}: {e}")
    return []

```

Cette fonction:

- Récupère les étapes détaillées d'un guide depuis l'API iFixit en utilisant son ID
- Extrait et structure les données textuelles des étapes
- Gère les erreurs et les cas exceptionnels

utils.py

Ce module fournit diverses fonctions utilitaires pour le traitement et le formatage des données.

Fonctions principales:

format_documents()

```

def format_documents(docs):
    """Formate les documents pour l'affichage"""
    from api_client import get_guide_steps

    formatted_docs = []
    for doc in docs:
        guide_id = doc.metadata.get("guideid")
        if guide_id:
            guide_steps = get_guide_steps(guide_id)
            guide_infos = ""
            for guide in guide_steps:
                step_text = "\n".join(guide["text"])
                guide_infos += "\n" + f"Step {guide['stepno']}: \n" + step_text

            if guide_infos not in doc.page_content:
                doc.page_content += guide_infos

    metadata_text = "\n".join(f"{key}: {value}" for key, value in

```

```

doc.metadata.items()
    formatted_doc = f"""---\n📄 Contenu :\n{doc.page_content}\n\n📄
Métadonnées :\n{doc.metadata_text}\n"""
    formatted_docs.append(formatted_doc)

return "\n\n".join(formatted_docs)

```

Cette fonction:

- Enrichit les documents guides avec leurs étapes détaillées (requêtes API)
- Formate les documents pour l'affichage avec séparation contenu/métadonnées
- Prépare les documents pour être utilisés dans le contexte du LLM

get_unique_union()

```

def get_unique_union(documents: list[list[Document]]) -> list[Document]:
    """Renvoie une liste de documents uniques à partir d'une liste de listes de
    documents."""
    # Aplatir la liste
    flattened_docs = [
        dumps(doc.__dict__, sort_keys=True) for sublist in documents for doc in
        sublist
    ]

    # Supprimer les doublons
    unique_docs = list(set(flattened_docs))

    # Reconvertir en objets Document
    return [Document(**loads(doc)) for doc in unique_docs]

```

Cette fonction:

- Prend plusieurs listes de documents (résultats de plusieurs requêtes)
- Élimine les doublons pour éviter la redondance
- Retourne une liste unifiée de documents uniques

templates/index.html

Ce fichier définit l'interface utilisateur principale de l'application, présentant un design moderne avec une structure à deux panneaux.

Structure principale:

```

<!DOCTYPE html>
<html lang="fr">
  <head>
    <!-- Métadonnées et liens vers CSS/JS externes -->
  </head>

```

```

<body>
  <div class="app-container">
    <!-- Sidebar -->
    <div class="sidebar bg-white">
      <!-- Logo et titre -->
      <div class="p-4">
        <div class="flex items-center mb-6">
          <!-- Logo et titre de l'application -->
        </div>

        <!-- Indicateur de temps de traitement -->
        <div class="flex items-center bg-indigo-50 py-2 px-3 rounded-lg mb-4">
          <!-- Affichage du temps de traitement -->
        </div>

        <!-- Boutons de toggle pour les requêtes et documents -->
        <div class="flex space-x-2 mb-4">
          <!-- Boutons pour montrer/cacher les panneaux -->
        </div>

        <!-- Conteneur pour les requêtes alternatives -->
        <div id="queries-container" class="queries-container mb-4 bg-indigo-50
rounded-xl p-3 border border-indigo-100">
          <!-- Liste des requêtes générées -->
        </div>

        <!-- Conteneur pour les documents récupérés -->
        <div id="documents-container" class="documents-container bg-blue-50
rounded-xl p-3 border border-blue-100">
          <!-- Liste des documents utilisés pour la réponse -->
        </div>
      </div>

      <!-- Footer de la sidebar -->
      <div class="mt-auto p-4 text-center text-gray-500 text-xs">
        <!-- Informations de copyright et technologie -->
      </div>
    </div>

    <!-- Contenu principal - Chat -->
    <div class="main-content">
      <!-- En-tête du chat -->
      <div class="chat-header">
        <!-- Bouton toggle sidebar et indicateur de statut -->
      </div>

      <!-- Zone de chat avec défilement -->
      <div class="chat-container bg-gray-50">
        <!-- Messages -->
        <div id="chat-messages" class="chat-messages">
          <!-- Message de bienvenue et historique des messages -->
        </div>

        <!-- Zone de saisie fixe en bas -->

```

```
<div class="chat-input-container">
  <!-- Champ de texte et bouton d'envoi -->
</div>
</div>
</div>
</div>
</body>
</html>
```

Caractéristiques principales:

Pour rendre le chatbot à la fois lisible et transparent quant à son fonctionnement interne, j'ai opté pour une interface organisée en deux volets:

1. Double panneau

- Un panneau latéral (sidebar) affiche les métriques et les données propres au RAG.
- Le panneau principal héberge la zone de conversation.
- Cette disposition permet de suivre le dialogue tout en gardant un œil sur le «backstage» du système.

2. Sidebar informative

- Indicateur du temps de traitement de la requête.
- Sections repliables détaillant les requêtes alternatives générées.
- Affichage des documents récupérés, accompagnés de leurs contenus et métadonnées.

3. Zone de chat moderne

- Fil de messages avec rendu Markdown pour un affichage propre des listes, liens ou extraits de code.
- Barre de saisie fixée en bas de l'écran pour éviter le défilement intempestif.

4. Éléments interactifs

- Bouton pour masquer ou afficher la sidebar.
- Toggles permettant d'ouvrir ou de fermer les sections «Requêtes» et «Documents»
- Tableau structuré listant les sources utilisées, afin de favoriser la traçabilité.

5. Dépendances externes

- Font Awesome pour les icônes.
- Tailwind CSS pour un style responsive sans alourdir le code.
- Marked.js pour interpréter le Markdown dans les messages.
- DOMPurify pour sécuriser le rendu HTML et éviter toute injection malveillante.

Cette organisation vise à concilier une expérience de chat fluide et l'exigence de transparence : à tout moment, l'utilisateur peut vérifier d'où vient l'information et comment la réponse a été élaborée.

static/css/style.css

Ce fichier contient les styles personnalisés pour l'application, complétant Tailwind CSS avec des animations et des styles spécifiques.

Catégories de styles:

1. Styles de base:

```
body {
  font-family: 'Poppins', sans-serif;
  color: #333;
  line-height: 1.6;
  height: 100vh;
  overflow: hidden;
}
```

Ces styles de base fixent la police principale (Poppins) et la couleur du texte de l'application. Avec la propriété `height: 100vh`, l'application remplit toute la hauteur de l'écran. Le `overflow: hidden` évite le défilement, ce qui donne une expérience plus proche d'une application que d'un site web classique.

2. Animations:

```
/* Animation de chargement */
.loading-dots:after {
  content: '';
  animation: dots 1.5s steps(5, end) infinite;
}

@keyframes dots {
  0%, 20% { content: '.'; }
  40% { content: '..'; }
  60% { content: '...'; }
  80%, 100% { content: ''; }
}

/* Animation de pulsation */
.pulse-animation {
  animation: pulse 2s infinite;
}

@keyframes pulse {
  0% { box-shadow: 0 0 0 0 rgba(72, 187, 120, 0.7); }
  70% { box-shadow: 0 0 0 10px rgba(72, 187, 120, 0); }
  100% { box-shadow: 0 0 0 0 rgba(72, 187, 120, 0); }
}
```

Cette partie parle de deux animations principales pour l'interface. D'abord, il y a un effet de chargement qui montre que le chatbot est en train de faire quelque chose. Ensuite, il y a une animation de pulsation qui fait vibrer certains éléments pour capter l'attention de l'utilisateur. Ces animations ajoutent un peu de dynamisme à l'interface et aident à comprendre ce qui se passe.

3. Layout et structure de l'application:

```
.app-container {
  display: flex;
  height: 100vh;
```

```
    width: 100%;
  }

  .sidebar {
    width: 280px;
    flex-shrink: 0;
    height: 100vh;
    overflow-y: auto;
    transition: transform 0.3s ease;
    border-right: 1px solid #e2e8f0;
  }

  .main-content {
    flex-grow: 1;
    height: 100vh;
    display: flex;
    flex-direction: column;
  }
}
```

Dans cette section, on va discuter de l'organisation de l'application avec **Flexbox**. Le conteneur principal, qu'on appelle **app-container**, est divisé en deux parties : une sidebar qui fait 280px de large et une zone de contenu qui remplit le reste de l'espace. Si le contenu de la sidebar dépasse, on peut la faire défiler, et elle s'ouvre et se ferme avec une animation fluide.

4. Gestion de la sidebar:

```
.toggle-sidebar {
  position: static;
  border: none;
  background: transparent;
  color: white;
  display: flex;
  align-items: center;
  justify-content: center;
  width: 32px;
  height: 32px;
  border-radius: 8px;
  transition: all 0.2s ease;
}

.sidebar-collapsed .sidebar {
  margin-left: -280px;
}

.sidebar-collapsed .chat-input-container {
  left: 0;
}
```

Ces styles gèrent le comportement de la sidebar, notamment sa capacité à se réduire et s'étendre. Le bouton **toggle-sidebar** est stylisé pour s'intégrer harmonieusement à l'en-tête, avec un effet de survol subtil.

Lorsque la classe `sidebar-collapsed` est appliquée à l'élément parent, la sidebar se déplace hors de l'écran grâce à une marge négative, et la zone de saisie du chat s'ajuste automatiquement pour occuper toute la largeur disponible, optimisant ainsi l'espace d'affichage.

5. Interface de chat:

```
.chat-container {
  display: flex;
  flex-direction: column;
  flex: 1;
  overflow: hidden;
}

.chat-header {
  background: linear-gradient(to right, #4f46e5, #3b82f6);
  padding: 0.75rem 1rem;
  display: flex;
  align-items: center;
  justify-content: space-between;
  color: white;
}

.chat-messages {
  flex: 1;
  overflow-y: auto;
  scrollbar-width: thin;
  scrollbar-color: rgba(79, 70, 229, 0.2) transparent;
  padding: 1rem;
  padding-bottom: 80px;
}

.chat-input-container {
  position: fixed;
  bottom: 0;
  left: 0;
  right: 0;
  padding: 1rem;
  background-color: white;
  border-top: 1px solid #e2e8f0;
  z-index: 10;
  box-shadow: 0 -2px 10px rgba(0, 0, 0, 0.05);
  transition: all 0.3s ease;
}
```

Cette partie parle de l'apparence et du fonctionnement de l'interface de chat. L'en-tête utilise un dégradé de couleurs indigo/bleu pour un effet visuel attrayant. La zone de messages est configurée pour défiler automatiquement, avec des barres de défilement stylisées et discrètes. Un espace supplémentaire est ajouté en bas (`padding-bottom: 80px`) pour éviter que les derniers messages ne soient masqués par la zone de saisie. La zone de saisie est fixée en bas de l'écran avec un effet d'ombre subtil, garantissant qu'elle reste toujours accessible pendant le défilement de la conversation.

6. Styles des messages:

```
.message {
  max-width: 85%;
  margin-bottom: 20px;
  display: flex;
  position: relative;
  animation: fadeIn 0.3s ease-out;
}

@keyframes fadeIn {
  from {
    opacity: 0;
    transform: translateY(5px);
  }
  to {
    opacity: 1;
    transform: translateY(0);
  }
}

.message-avatar {
  width: 36px;
  height: 36px;
  border-radius: 50%;
  display: flex;
  align-items: center;
  justify-content: center;
  margin-right: 12px;
  flex-shrink: 0;
}

.message-bot .message-avatar {
  background: linear-gradient(45deg, #4f46e5, #3b82f6);
  color: white;
  box-shadow: 0 4px 6px rgba(79, 70, 229, 0.15);
}

.message-user .message-avatar {
  background: linear-gradient(45deg, #10b981, #059669);
  color: white;
  box-shadow: 0 4px 6px rgba(16, 185, 129, 0.15);
}

.message-content {
  background-color: #fff;
  padding: 12px 16px;
  border-radius: 16px;
  box-shadow: 0 2px 5px rgba(0, 0, 0, 0.05);
  position: relative;
}

.message-user {
```

```
margin-left: auto;
flex-direction: row-reverse;
}

.message-user .message-content {
background-color: #ecfdf5;
border-top-right-radius: 4px;
border-right: 1px solid rgba(16, 185, 129, 0.1);
border-top: 1px solid rgba(16, 185, 129, 0.1);
}
```

Cette section parle de l'apparence et du fonctionnement de l'interface de chat. L'en-tête a un joli dégradé de couleurs indigo et bleu qui attire l'œil. La zone des messages fait défiler automatiquement les nouvelles entrées, avec des barres de défilement discrètes. On explique aussi comment les messages sont présentés dans la conversation. Chaque message entre avec une animation fluide, ce qui rend les échanges plus dynamiques. Les avatars sont colorés différemment : l'utilisateur en vert et le bot en indigo/bleu. Les messages de l'utilisateur sont à droite avec un fond légèrement coloré, tandis que ceux du bot sont à gauche avec un fond blanc. Des petits détails comme des bords asymétriques et des ombres légères ajoutent un peu de style à l'interface.

7. Affichage des requêtes et documents:

```
.queries-container,
.documents-container {
display: none;
transition: all 0.3s ease;
box-shadow: inset 0 2px 4px rgba(0, 0, 0, 0.05);
max-height: 300px;
overflow-y: auto;
overflow-x: hidden;
word-wrap: break-word;
word-break: break-word;
}

.document-item {
overflow-wrap: break-word;
word-wrap: break-word;
word-break: break-word;
hyphens: auto;
}

.document-content,
.document-full-content {
overflow-wrap: break-word;
word-wrap: break-word;
word-break: break-word;
hyphens: auto;
max-width: 100%;
}
```

Ces styles gèrent les panneaux pliables dans la barre latérale, affichant des requêtes alternatives et des documents source. Par défaut, ces panneaux sont cachés et apparaissent avec une transition fluide quand on les active. On fixe une hauteur maximale et un défilement vertical pour bien contenir beaucoup d'informations sans déranger la mise en page. Diverses options de gestion du texte s'assurent que les mots longs et les URLs restent dans les conteneurs, ce qui aide à garder une présentation claire des documents et requêtes.

8. Rendu du Markdown:

```
.markdown-body h1 {
  font-size: 1.75rem;
  font-weight: 600;
  margin-top: 1.5rem;
  margin-bottom: 0.75rem;
  color: #1e293b;
}

.markdown-body h2 {
  font-size: 1.5rem;
  font-weight: 600;
  margin-top: 1.5rem;
  margin-bottom: 0.75rem;
  color: #1e293b;
}

.markdown-body code {
  font-family: 'Menlo', 'Monaco', 'Courier New', monospace;
  background-color: #f1f5f9;
  padding: 0.2rem 0.4rem;
  border-radius: 0.25rem;
  font-size: 0.875rem;
  color: #4f46e5;
  border: 1px solid #e2e8f0;
}
```

Cette section explique comment le formatage Markdown est utilisé dans les réponses du chatbot. Les titres ont des tailles et des poids différents pour que ce soit plus facile à lire. Les blocs de code sont en monospace avec un fond gris clair et une bordure discrète, ce qui rend le code facile à repérer. La couleur indigo du texte de code s'accorde avec le thème de l'application. Ces styles aident à rendre les réponses avec du texte formaté plus claires et pro.

9. Sections spécifiques et composants visuels:

```
.problem-analysis,
.checks,
.procedure,
.tips,
.sources {
  margin: 1.25rem 0;
  padding: 1rem;
```

```

    border-radius: 0.5rem;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.05);
    position: relative;
    padding-left: 1.25rem;
}

.problem-analysis {
    background-color: #eef2ff;
    border-left: 4px solid #4f46e5;
}

.checks {
    background-color: #ecfdf5;
    border-left: 4px solid #10b981;
}

.procedure {
    background-color: #fff7ed;
    border-left: 4px solid #f97316;
}

.tips {
    background-color: #fff6ff;
    border-left: 4px solid #d81b60;
}

.sources {
    background-color: #f8f9fc;
    border-left: 4px solid #4a5568;
}

```

Ces styles aident à distinguer les différentes informations dans les réponses du chatbot. Chaque partie, comme l'analyse de problème, les vérifications, les procédures, les conseils et les sources, a sa propre couleur avec un fond clair et une bordure à gauche. Ça rend les réponses plus lisibles et permet aux utilisateurs de trouver rapidement les informations qu'ils cherchent, même quand c'est un peu compliqué. Les ombres légères et les coins arrondis apportent une petite touche sympa, en séparant bien les sections.

10. Responsive design:

```

@media (max-width: 768px) {
    .sidebar {
        transform: translateX(-100%);
    }

    .message {
        max-width: 95%;
    }

    .chat-input-container {
        left: 0;
        padding: 0.75rem;
    }
}

```

```

    }

    .input-wrapper {
      width: 100%;
      max-width: 100%;
      display: flex;
      align-items: center;
    }
  }
}

```

Cette partie s'assure que l'interface fonctionne bien sur les mobiles et les écrans plus petits. Sur les écrans étroits (jusqu'à 768px), la barre latérale se cache automatiquement pour laisser plus de place aux messages. Ces derniers occupent une plus grande largeur (95% au lieu de 85%), et la zone de saisie est modifiée avec moins de marge pour que ce soit facile à utiliser sur les petits écrans. L'input-wrapper s'étend sur toute la largeur et garde les éléments bien alignés. Tous ces réglages permettent d'avoir une bonne expérience, que l'on soit sur un ordinateur ou un smartphone.

static/js/main.js

Ce module JavaScript gère l'interface utilisateur côté client, les interactions et les animations du chatbot.

Fonctions principales:

Initialisation et configuration

```

document.addEventListener('DOMContentLoaded', function () {
  // Récupération des éléments DOM
  const chatMessages = document.getElementById('chat-messages')
  const userInput = document.getElementById('user-input')
  const sendButton = document.getElementById('send-button')
  // ...
})

```

Le code s'initialise au chargement complet du DOM et configure les références vers les éléments d'interface.

formatMarkdown()

```

function formatMarkdown(text) {
  // Formatage spécial pour les sections du format de réponse attendu
  text = text.replace(/🔍\s*\*\*Analyse du problème\*\*\s*/g, '<div class="problem-analysis"><strong>🔍 Analyse du problème :</strong>')
  // Autres remplacements...

  // Utiliser marked pour le reste du formatage markdown
  let formattedText = marked.parse(text)

  // Sanitiser le HTML pour éviter les injections XSS

```

```
    return DOMPurify.sanitize(formattedText)
}
```

Cette fonction convertit le texte markdown en HTML avec des améliorations visuelles pour les sections spécifiques des réponses du chatbot. Elle utilise la bibliothèque `marked.js` et `DOMPurify` pour la sanitisation du contenu.

addMessage()

```
function addMessage(content, isUser = false) {
    const messageDiv = document.createElement('div')
    messageDiv.className = `message ${isUser ? 'message-user' : 'message-bot'}`

    // Création de l'avatar
    const avatarDiv = document.createElement('div')
    avatarDiv.className = 'message-avatar'
    // ...

    // Animation d'entrée
    setTimeout(() => {
        messageDiv.style.opacity = '1'
        messageDiv.style.transform = 'translateY(0)'
    }, 10)

    return { messageDiv, innerDiv }
}
```

Ajoute un nouveau message dans la conversation avec des styles différents selon qu'il provient de l'utilisateur ou du chatbot. Inclut des animations pour une expérience utilisateur fluide.

addTypingEffect()

```
async function addTypingEffect(element, content) {
    // Paramètres ajustés pour un effet visuel plus naturel
    const minDelay = 10; // Délai minimum entre les caractères (ms)
    const maxDelay = 25; // Délai maximum entre les caractères (ms)
    const chunkSize = 3; // Nombre de caractères à traiter par itération
    const fastModeThreshold = 800; // Seuil à partir duquel on accélère le traitement

    // Mode accéléré pour les contenus longs
    const fastMode = content.length > fastModeThreshold;

    // Pour les longs textes ou le mode rapide, traitement par chunks plus grands
    if (fastMode) {
        for (let i = 0; i < content.length; i += chunkSize * 2) {
            const endPos = Math.min(i + chunkSize * 2, content.length);
            currentText += content.substring(i, endPos);
        }
    }
}
```

```

        element.innerHTML = formatMarkdown(currentText);
        chatMessages.scrollTop = chatMessages.scrollHeight;

        await new Promise(resolve => setTimeout(resolve, 5));
    }
} else {
    // Mode normal avec effet de frappe visible
    for (let i = 0; i < content.length; i += chunkSize) {
        const endPos = Math.min(i + chunkSize, content.length);
        currentText += content.substring(i, endPos);
        element.innerHTML = formatMarkdown(currentText);
        chatMessages.scrollTop = chatMessages.scrollHeight;

        await new Promise(resolve => setTimeout(resolve,
            Math.floor(Math.random() * (maxDelay - minDelay + 1)) +
minDelay));
    }
}
}

```

Cette fonctionnalité fait que le texte du chatbot s'affiche comme s'il était écrit en direct. Elle change la vitesse en fonction de la longueur du message, allant plus vite pour les réponses longues et ralentissant pour les plus courtes, en montrant le texte lettre par lettre avec des pauses différentes.

addLoadingMessage() et removeLoadingMessage()

```

function addLoadingMessage() {
    const loadingDiv = document.createElement('div')
    // ...
    innerDiv.innerHTML = `
        <div class="flex items-center space-x-2">
            <svg class="animate-spin h-4 w-4 text-indigo-600"
xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24">
                <!-- ... -->
            </svg>
            <span class="text-indigo-700 font-medium">Analyse en cours</span>
            <span class="loading-dots text-indigo-700"></span>
        </div>
    `
    // ...
}

```

```

// Fonction pour supprimer le message de chargement
function removeLoadingMessage() {
    const loadingMessage = document.getElementById('loading-message')
    if (loadingMessage) {
        loadingMessage.remove()
    }
}

```

```

    }
  }
}

```

Affiche et supprime un indicateur animé de chargement pendant le traitement des requêtes.

displayQueries() et gestion du toggle

```

function displayQueries(queries) {
  queriesList.innerHTML = ''

  // Ajouter les requêtes avec une légère animation
  queries.forEach((query, index) => {
    setTimeout(() => {
      // Création et animation des éléments
    }, index * 100) // Ajouter un délai pour chaque élément
  })

  // Animation du conteneur
  // ...
}

// Gestion du toggle pour les requêtes avec animation
toggleQueries.addEventListener('click', function () {
  // Animation d'ouverture/fermeture
})

```

Affiche les requêtes alternatives générées par le système avec des animations élégantes. Le toggle permet d'afficher/masquer cette section.

displayDocuments(), formatDocumentContent() et gestion du toggle

```

function displayDocuments(documents) {
  // Affichage des documents récupérés avec animations
  // ...
}

function formatDocumentContent(docText) {
  // Extraction et formatage du contenu et des métadonnées
  // ...
  return `
    <div class="document-header flex items-center justify-between mb-2">
      <!-- Template du document formaté -->
    </div>
    <!-- ... -->
  `
}

// Gestion du toggle pour les documents
toggleDocuments.addEventListener('click', function () {

```



```

    // Animation d'ouverture/fermeture
  })

  // Délégation d'événements pour les boutons "Voir détails"
  documentsList.addEventListener('click', function (e) {
    if (e.target.classList.contains('view-more-btn') ||
    e.target.parentElement.classList.contains('view-more-btn')) {
      const button = e.target.classList.contains('view-more-btn') ? e.target :
      e.target.parentElement
      const fullContent = button.nextElementSibling

      if (fullContent.classList.contains('hidden')) {
        fullContent.classList.remove('hidden')
        button.innerHTML = '<i class="fas fa-eye-slash mr-1"></i> Masquer
détails'
      } else {
        fullContent.classList.add('hidden')
        button.innerHTML = '<i class="fas fa-eye mr-1"></i> Voir détails'
      }
    }
  })
})

```

Ces fonctions gèrent l'affichage, le formatage et les interactions avec la section des documents récupérés. Inclut des fonctionnalités comme l'affichage/masquage des détails et des animations fluides.

sendMessage()

```

async function sendMessage() {
  const message = userInput.value.trim()
  if (!message) return

  // Désactiver l'input pendant l'envoi
  userInput.disabled = true
  // ...

  try {
    // Appel API avec fetch
    const response = await fetch('/api/chat', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({ message })
    })

    const data = await response.json()

    // Traitement de la réponse
    // ...
  } catch (error) {
    // Gestion des erreurs
  }
}

```

```
    } finally {  
        // Réactivation de l'interface  
    }  
}
```

Fonction centrale qui envoie les messages utilisateur à l'API, affiche les réponses et met à jour l'interface utilisateur en conséquence. Gère l'état de l'interface pendant les communications avec le serveur.

Gestion responsive et toggle de la sidebar

```
// Vérifier si on est en mode mobile au chargement  
function checkMobileView() {  
    if (window.innerWidth <= 768) {  
        appContainer.classList.add('sidebar-collapsed');  
  
        // Mettre à jour l'icône  
        const icon = sidebarToggle.querySelector('i');  
        icon.classList.remove('fa-chevron-left');  
        icon.classList.add('fa-chevron-right');  
  
        // S'assurer que la sidebar est bien masquée  
        sidebar.style.transform = 'translateX(-100%)';  
    } else {  
        // En mode desktop, s'assurer que la sidebar est visible  
        sidebar.style.transform = 'translateX(0)';  
    }  
}  
  
// Toggle sidebar avec correction pour mobile  
sidebarToggle.addEventListener('click', function () {  
    appContainer.classList.toggle('sidebar-collapsed');  
    const icon = sidebarToggle.querySelector('i');  
  
    if (appContainer.classList.contains('sidebar-collapsed')) {  
        icon.classList.remove('fa-chevron-left');  
        icon.classList.add('fa-chevron-right');  
        // Animation fluide pour la sidebar  
        sidebar.style.transform = 'translateX(-100%)';  
    } else {  
        icon.classList.remove('fa-chevron-right');  
        icon.classList.add('fa-chevron-left');  
        // Animation fluide pour la sidebar  
        sidebar.style.transform = 'translateX(0)';  
    }  
  
    // Ajuster la position du conteneur d'input  
    const chatInputContainer = document.querySelector('.chat-input-container');  
    if (appContainer.classList.contains('sidebar-collapsed')) {  
        chatInputContainer.style.left = '0';  
    } else {  
        if (window.innerWidth > 768) {
```

```
        chatInputContainer.style.left = '280px';
    } else {
        chatInputContainer.style.left = '0';
    }
}
});
```

Gère comment l'application s'affiche sur différents écrans. Sur un mobile, elle adapte la barre latérale. Le système regarde la taille de l'écran quand la page se charge et aussi quand elle se redimensionne pour que ça fonctionne bien autant sur mobile que sur ordinateur.

Caractéristiques notables:

Pour rendre le chatbot agréable à utiliser tout en restant transparent sur son fonctionnement interne, j'ai ajouté plusieurs raffinements :

1. **Animations douces** : l'effet de frappe simule une réponse qui se dévoile petit à petit, tandis que les volets latéraux s'ouvrent et se referment sans à-coups. Le tout crée une impression de fluidité qui rappelle les messageries instantanées.
2. **Affichage malin du contenu** : le Markdown est pré-traité pour éviter les surprises d'affichage ; la vitesse de défilement s'adapte à la longueur du texte ; et les sections importantes (code, listes, blocs de citation) bénéficient d'un style un peu plus soigné pour ressortir au premier coup d'œil.
3. **Confort visuel et retours d'état** : un petit indicateur animé signale que la requête est en cours de traitement. Dès que la réponse arrive, un changement de couleur rappelle le temps mis par le système, ce qui aide à évaluer les performances en direct.
4. **Sécurité en tête** : chaque fois qu'un bloc HTML doit être injecté, DOMPurify passe avant pour filtrer les balises douteuses. De plus, j'ai ajouté un contrôle simple sur les saisies utilisateur pour éviter des injections involontaires. Si l'API tombe, le message d'erreur reste lisible et sans jargon technique.
5. **Vue détaillée des sources** : un clic sur un document ouvre un encart qui affiche le passage exact, ainsi que ses métadonnées. On sait immédiatement d'où vient chaque information.
6. **Responsive par nature** : l'interface détecte les écrans mobiles et réorganise automatiquement les panneaux. Sur téléphone, la sidebar se replie pour laisser toute la place au fil de discussion, mais reste accessible via un bouton flottant.

Scripts de collecte de données

get_posts.py

Afin d'alimenter la base de connaissances avec des retours d'expérience concrets, j'ai rédigé un petit script qui va puiser dans le subreddit [r/techsupport](#). Voici, en termes simples, ce qu'il fait :

1. **Connexion** : le script s'identifie auprès de Reddit via [PRAW](#) à l'aide de mes identifiants d'API.
2. **Parcours des posts** : il parcourt la liste des sujets récents.
3. **Extraction des discussions** : pour chaque post, il retient une vingtaine de commentaires parmi les plus pertinents.
4. **Sauvegarde** : l'ensemble est stocké dans un fichier JSON, prêt à être nettoyé puis indexé dans FAISS.

Ce petit outil me permet d'actualiser la base quand je le souhaite, sans devoir passer par une collecte manuelle fastidieuse.

Vue d'ensemble

Ce script récupère les publications et jusqu'à vingt commentaires par post sur le subreddit [r/techsupport](#) en s'appuyant sur l'API [PRAW](#), puis stocke le tout dans un fichier JSON.

Étapes principales :

1. Connexion à Reddit via [PRAW](#) à l'aide des identifiants fournis.
2. Parcours des posts récents du subreddit techsupport.
3. Extraction de chaque post, accompagnée d'un maximum de 20 commentaires.
4. Sérialisation des données (titre, corps du message, commentaires, ...) dans un fichier JSON pour un traitement ultérieur (indexation).

Configuration

```
# Connexion à l'API Reddit
reddit = praw.Reddit(
    client_id="6NAkftfhMW0UzBTsOk_WUw",
    client_secret="DGo0-wC2rWF9kuHRk8EcvQ5IIvoq2Q",
    user_agent="script:reddit.techsupport.scraper:v1.0 (par
/u/Ambitious_Shopping45)",
)

# Fichier JSON pour stocker les posts avec leurs commentaires
FICHIER_JSON = "techsupport_posts.json"
```

La configuration inclut les identifiants d'API nécessaires et le chemin du fichier de sortie.

Fonctions principales

Récupération et traitement de posts

```
# Récupérer un maximum de posts
posts_data = []

for post in subreddit.top(limit=None): # Pas de limite de posts
    post.comments.replace_more(limit=0) # Supprime les "More Comments"

    # Récupérer jusqu'à 20 commentaires par post
    comments = [comment.body for comment in post.comments.list()[:20]]

    post_info = {
        "id": post.id,
        "titre": post.title,
        "contenu": post.selftext,
```

```
        "url": post.url,
        "date": post.created_utc,
        "score": post.score,
        "nombre_commentaires": post.num_comments,
        "comments": comments, # Liste des 20 premiers commentaires max
    }

    posts_data.append(post_info)
```

Cette section:

- Parcourt les posts du subreddit en ordre de popularité (pas de limite)
- Pour chaque post, extrait jusqu'à 20 commentaires
- Structure les données de post et commentaires dans un dictionnaire
- Ajoute chaque post traité à une liste

Sauvegarde des données

```
# Sauvegarde en une seule fois (plus rapide)
with open(FICHIER_JSON, "w", encoding="utf-8") as f:
    json.dump(posts_data, f, indent=4, ensure_ascii=False)

print(
    f"{len(posts_data)} posts enregistrés avec un max de 20 commentaires chacun
    dans {FICHIER_JSON} !"
)
```

Cette section:

- Sauvegarde tous les posts collectés dans un fichier JSON
- Utilise un encodage UTF-8 pour préserver les caractères spéciaux
- Affiche un message de confirmation avec le nombre de posts récupérés

guides.py

Ce module est responsable de récupérer les guides techniques depuis l'API publique de iFixit.

Vue d'ensemble

Le script se connecte à l'API iFixit, récupère tous les guides disponibles via une pagination, et sauvegarde l'ensemble dans un fichier JSON.

Fonctions principales

`fetch_all_guides()`

```
def fetch_all_guides():
    all_guides = []
    offset = 0
    limit = 20
    has_more = True

    while has_more:
        url = f"https://www.ifixit.com/api/2.0/guides?offset={offset}&limit={limit}"
        response = requests.get(url)

        # Vérifiez la réponse
        print(f"Status Code: {response.status_code}")
        print(
            f"Response Content: {response.content[:200]}..."
        ) # Affiche les premiers 200 caractères de la réponse

        if response.status_code == 200:
            try:
                guides = response.json()
                all_guides.extend(guides)

                if len(guides) < limit:
                    has_more = False
                else:
                    offset += limit
            except ValueError:
                print("Erreur de décodage JSON")
                break
        elif response.status_code == 429:
            # Si la limite est atteinte, attendons 30 secondes avant de continuer
            print("Limite de requêtes atteinte. Attente de 30 secondes...")
            time.sleep(30) # Attendre 30 secondes avant de réessayer
        else:
            print(
                f"Erreur lors de la récupération des données. Code statut: {response.status_code}"
            )
            break

    return all_guides
```

Cette fonction fait plusieurs choses :

- Elle récupère tous les guides, en prenant 20 à la fois.
- Si l'API atteint sa limite (code 429), elle attend automatiquement avant de continuer.
- Elle montre des informations de débogage sur le statut de chaque requête.
- Tous les guides sont mis dans une liste.

Exécution principale

```
# Récupérer tous les guides
guides = fetch_all_guides()

# Enregistrement des guides dans un fichier JSON
if guides:
    with open("./data/guides.json", "w") as f:
        json.dump(guides, f, indent=2)
    print("Données enregistrées dans guides.json")
else:
    print("Aucun guide trouvé ou problème avec l'API.")
```

Cette section:

- Appelle la fonction pour récupérer tous les guides
- Vérifie que des guides ont été trouvés
- Sauvegarde les guides dans un fichier JSON
- Affiche un message de confirmation ou d'erreur selon le résultat

Flux de données

Pour clarifier le chemin parcouru par l'information dans le système, j'ai résumé le processus en cinq étapes clés :

1. **Saisie de la question:**
 - L'utilisateur soumet sa requête depuis l'interface web ou via l'API
2. **Préparation des requêtes:**
 - Le moteur crée cinq reformulations différentes grâce à la chaîne `generate_queries`
3. **Récupération:**
 - Chaque variante interroge l'index FAISS ; les résultats sont fusionnés et dé-dupliqués (via `get_unique_union`), puis enrichis (le cas échéant) par les étapes détaillées récupérées sur iFixit
4. **Génération de réponse:**
 - Les passages retenus sont insérés dans le contexte et transmis à GPT-4.1, qui rédige une réponse structurée selon le template prévu
5. **Retour à l'utilisateur:**
 - La réponse, les requêtes générées, les documents utilisés et le temps de traitement sont renvoyés. Sur le front-end, ces données apparaissent dans la zone de chat et dans des volets repliables

Modèles et embeddings

Le pipeline repose sur trois briques principales :

1. **Modèle d'embeddings:**
 - J'utilise `sentence-transformers/all-MiniLM-L6-v2` (Hugging Face) pour convertir les textes en vecteurs, avec prise en charge de CUDA quand la carte graphique est disponible
2. **Modèle de langage:**
 - GPT-4.1 d'OpenAI, réglé sur une température de 0,0 afin de produire des réponses déterministes. Le même modèle génère aussi les requêtes alternatives

3. Base vectorielle:

- FAISS, configuré avec un `k` de 2 et un seuil de similarité de 0,3 pour équilibrer vitesse et précision.

Interface utilisateur

La majeure partie de la structure a déjà été décrite, mais voici un récapitulatif rapide :

Caractéristiques principales:

1. Disposition à deux panneaux:

- À gauche, une sidebar qui répertorie les métriques et les données RAG ; à droite, le fil de discussion. Sur mobile, la sidebar se masque automatiquement

2. Organisation visuelle:

- Messages utilisateur/bot bien différenciés, barre de saisie fixe, volets repliables pour consulter les requêtes et les documents

3. Visualisation des données RAG:

- Onglets pour les requêtes générées, tableau des documents, indicateur de temps de traitement

4. Fonctionnalités avancées:

- Tailwind CSS (style), Font Awesome (icônes), Marked.js (Markdown), DOMPurify (sanitisation)

5. Composants JavaScript:

- `main.js` gère les interactions utilisateur
- `marked.js` pour le rendu Markdown
- `DOMPurify` pour la sécurité du contenu HTML

Configuration et déploiement

Prérequis:

- Python 3.11
- Fichiers sources JSON (`./data/guides.json` et `./data/techsupport_posts.json`)
- Clé API OpenAI dans un fichier `.env`
- GPU compatible CUDA (optionnel mais recommandé)

Structure des fichiers:

```
.
├── api_client.py      # Client API pour iFixit
├── app.py             # Application Flask principale
├── rag_chain.py       # Configuration des chaînes RAG
├── requirements.txt   # Dépendances Python
├── retriever.py       # Gestion de la récupération des documents
├── utils.py           # Fonctions utilitaires
└── .env              # Variables d'environnement (OPENAI_KEY)
```



```
├── assets/                                # Autres ressources (images, etc.)
│   ├── architecture_diagram.svg          # Schéma de l'architecture de l'application
├── data/                                  # Données sources
│   ├── guides.json                       # Guides techniques
│   └── techsupport_posts.json            # Posts Reddit
├── static/                               # Ressources statiques
│   ├── css/
│   │   └── style.css                     # Styles CSS personnalisés
│   └── js/
│       └── main.js                       # Scripts JavaScript
└── templates/                            # Templates Flask
    └── index.html                        # Page d'accueil
```

Installation:

```
# Créer l'environnement virtuel
python -m venv venv

# Activer l'environnement virtuel
.\venv\Scripts\activate

# Si PowerShell bloque l'exécution de scripts :
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass

# Installer les dépendances
pip install -r requirements.txt

# Créer le fichier .env avec la clé API
echo "OPENAI_KEY=votre_cle_api" > .env

# Lancer l'application
python app.py
```

Points d'extension

Plusieurs pistes d'amélioration restent ouvertes; elles visent à renforcer la pertinence des réponses, la transparence du pipeline et le confort de l'utilisateur, sans remettre en question l'ossature du système.

1. Ajout de nouvelles sources:

- Modifier `retriever.py` pour intégrer de nouvelles sources de données
- Adapter le format des documents dans `index_data_embeddings()`

2. Changement de modèles:

- Modifier le modèle d'embeddings dans `index_data_embeddings()`
- Modifier le modèle LLM dans `initialize_rag_system()`

3. Optimisation des prompts:

- Adapter les templates dans `initialize_rag_system()`
- Affiner le format de réponse selon les besoins

4. Amélioration du retriever:

- Ajuster les paramètres du retriever (`chunk_size`, `search_kwargs`, etc.)
- Implémenter des techniques de ré-ordonnancement ou de boosting

5. **Interface utilisateur:**

- Modifier `templates/index.html` et `static/css/style.css` pour personnaliser l'interface
- Ajouter des fonctionnalités comme l'historique des conversations
- Intégrer des visualisations pour la pertinence des documents

6. **Optimisations frontend:**

- Améliorer les animations et transitions
- Ajouter des graphiques pour visualiser la pertinence des résultats
- Implémenter un mode sombre/clair
- Ajouter la sauvegarde de conversations

Chaque amélioration peut être intégrée de façon incrémentale; ainsi le cœur RAG demeure stable tandis que l'expérience globale gagne en richesse.