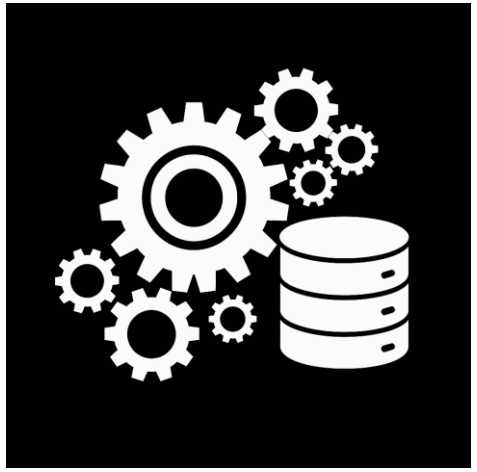



# IA & RAG
# Guillaume Oneill
# LockHeed LLC

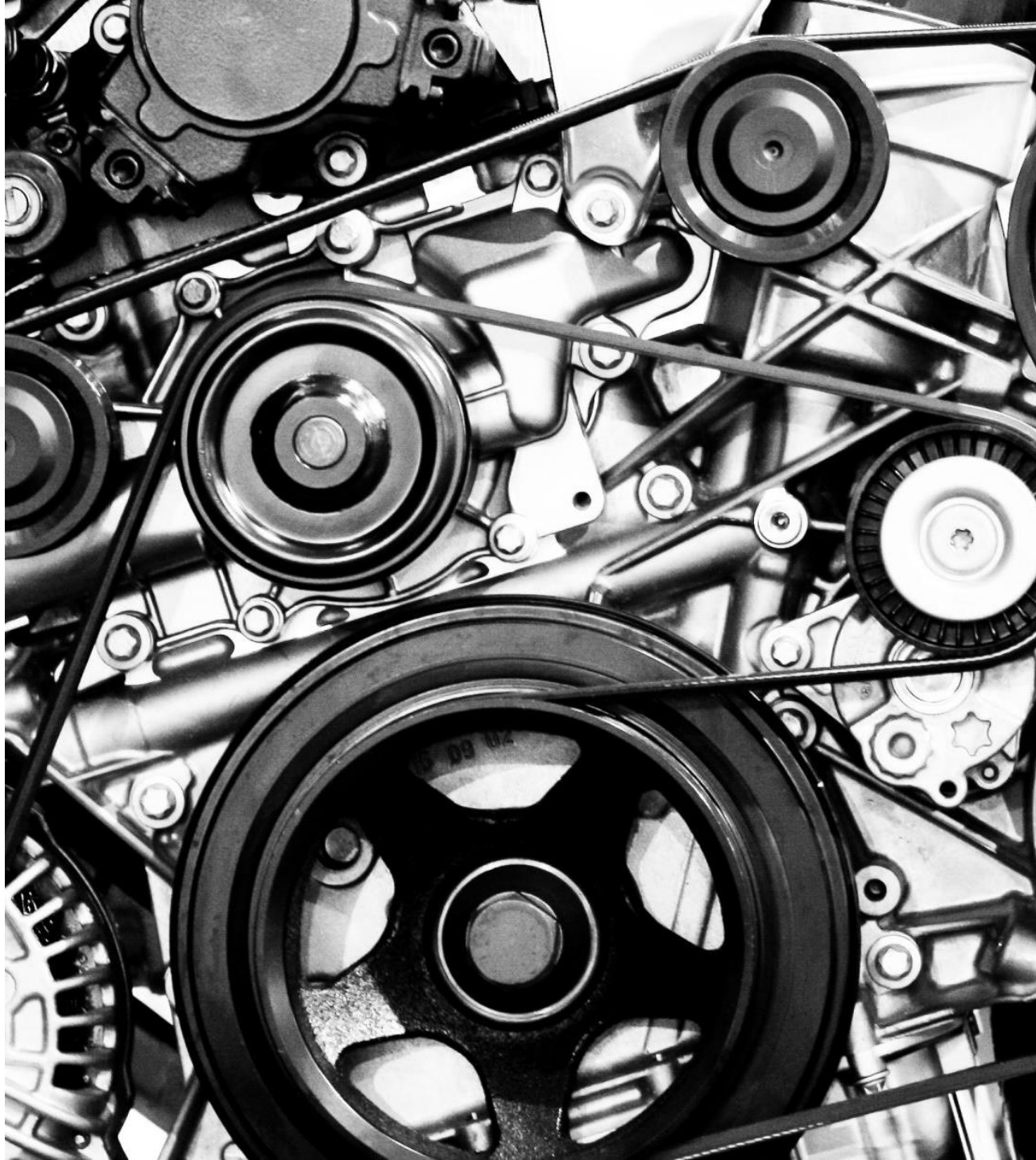

Ideo Lab

31th August 2025

Version 1.0

# AGENDA CONSEIL SUR l’IA ET LE RAG

- 1 Les moteurs d’IA disponibles



# A QUOI CELA SERT UN RAG

# 1) C’est quoi un RAG (Retrieval-Augmented Generation) ?

Un RAG combine:

- **Un moteur de recherche sémantique** (retrieval) sur tes données privées.
- **Un LLM génératif** qui **s’appuie** sur les passages retrouvés pour répondre.

  But: **réduire l’hallucination**, **actualiser** la connaissance sans ré-entraîner le modèle, **justifier** les réponses par des sources.

👉 **Un RAG, ça sert à donner à ChatGPT une "bibliothèque privée" qu'il peut consulter avant de répondre.**

Exemple concret :

- Tu demandes à ChatGPT : *"Donne-moi la meilleure recette de tarte aux pruneaux."*
- **Sans RAG** → ChatGPT se base uniquement sur ce qu'il a appris lors de son entraînement. Il peut inventer ou donner une recette générique.
- **Avec RAG** → Avant de répondre, ChatGPT va chercher dans **tes propres recettes enregistrées** (ta base de données, ton livre de cuisine, ton blog perso). Ensuite il mélange ça avec son savoir général et te répond **avec la bonne recette que tu avais chez toi**.

---

👉 Donc, un RAG = un **GPS pour l'IA** :

- Sans RAG → l'IA conduit "au feeling", parfois elle se trompe.
- Avec RAG → elle a la carte actualisée, et elle te montre le bon chemin avec des panneaux ("source : page 32 de ton livre de cuisine").

---

### 🌍 Situation sans RAG

Un dev demande à ChatGPT :

> "Comment construire un DataModel pour mon application Django ?"

👉 ChatGPT répond avec une solution **générique** : il parle de `models.py`, de relations `ForeignKey`, etc.
Mais le problème : ça ne tient **pas compte** du projet du dev (ses règles métiers, sa base existante, ses contraintes techniques).

---

### 📚 Situation avec RAG

Avec un **RAG**, avant de répondre :

1. L'IA va consulter la **documentation interne du projet** (les fichiers déjà codés, le schéma SQL, les guidelines de l'équipe, les tickets Jira, etc.).
2. Elle récupère les morceaux **les plus pertinents**.
3. Elle combine ces infos **avec son savoir général**.

Résultat :
👉 L'IA ne répond plus "n'importe quoi" → elle répond :

> "Vu ton fichier `models.py`, tu devrais ajouter une relation `OneToOne` avec ton modèle `UserProfile`. Attention, dans ton projet vous avez choisi `BigAutoField` comme clé primaire par défaut, donc il faut l'indiquer dans la migration."

### 🧠 Métaphore simple

Sans RAG, ChatGPT est comme **un prof qui connaît la théorie** mais ne connaît pas ton projet.
Avec RAG, c'est comme si ce prof **lisait tes cahiers et ton code** avant de t'expliquer la solution → donc la réponse est **personnalisée, exacte et directement applicable**.

---

👉 En résumé :

**Un RAG permet de transformer ChatGPT en un assistant "contextualisé" qui lit tes documents, ton code, ta base de données, et qui adapte ses réponses à TON environnement.**

---

# DESCRIPTIF D’UN RAG

## 2) Architecture de référence (production)

```
[Sources] → [Ingestion & Parsing] → [Chunking] → [Embeddings] → [Vector DB]
                    ↑                                   ↓
                (OCR, PDF,                      [Metadata enrich., BM25]
                 HTML, etc.)                            ↓
User Q → [Rewrite/Decompose] → [Retrieve (k)] → [Re-rank] → [Context Builder] → [LLM]
                                                    ↑
                                          [Policies/Guardrails]
                                                    ↓
                                          [Citations/Attributions]
                                                    ↓
                                          [Eval · Telemetry · Cache]
```

### Composants clés

- **Ingestion** : extracteurs fiables (PDF, HTML, DOCX), OCR si images.
- **Chunking** : découpe des documents (ex. 400–1 200 tokens, overlap 10–20%) *par structure* (titres, sections) plutôt que "bêtement".
- **Embeddings** : projection des chunks en vecteurs.
- **Index** : base vecteur (HNSW/IVF) + parfois index lexical (BM25) → **hybride** = mieux.
- **Retrieval** : top-k (ex. 10–20), filtres de métadonnées (langue, date, type doc).
- **Re-ranking** : reranker cross-encoder pour remonter les meilleurs 3–6 passages.
- **Prompting** : instructions + contexte + format réponse + citations.
- **Guardrails** : sécurité, PII, domaine autorisé, refus hors périmètre.
- **Observabilité** : logs, traces, coût, latence, qualité.
- **Cache** : réponses & embeddings (clé = (question normalisée, filtres, index-ver)).

## 3) Stratégies RAG qui comptent en entretien

- **Hybride lexical + vectoriel** (BM25 + ANN) → robustesse sur mots rares, codes, acronymes.
- **Re-ranking** (ex. bge-reranker, Cohere Rerank, Voyage-rerank) → bond de qualité.
- **Context Builder** soigné : *dedup*, *merge* par section, *budget de tokens*, *ordonnancement logique*.
- **Query rewriting :** réécrire la question (synonymes, expansion) ; décomposition *multi-hop* si nécessa
- **Routing** : router vers la bonne "collection" (connaissance produit vs politique RH).
- **Structured RAG** : extraire des **facts** structurés (table, JSON) avant la génération.
- **Freshness** : champ `last_updated` + filtres de date ; éventuellement **streaming updates**.
- **Eval continue** : jeux de questions, RAGAS/DeepEval, boucles d'amélioration.

# 4) Mise en œuvre (recette étape-par-étape)

## 4.1 Ingestion & parsing

- Parsers robustes: **Unstructured**, **Textract**, **Tika**; pour HTML: strip nav/ads, garder H1–H3.
- OCR: **Tesseract** ou services managés si documents scannés.
- Normalise **métadonnées**: source, date, auteur, type, langue, access_level.

## 4.2 Chunking (vraiment clé)

- Heuristique : *title-aware splitting* (couper aux titres, garder le titre dans chaque chunk).
- Taille: 600–1 000 tokens avec **overlap 100–200** si prose; plus petit pour code/FAQ.
- Fusion de petits paragraphes pour éviter les "micro-chunks" orphelins.

## 4.3 Embeddings & index

- Embeddings modernes (au choix) :
  - OpenAI `text-embedding-3-large` / `small`
  - BAAI **bge-m3** (multilingue), **e5-mistral**, **voyage-multilingual**
- Vector DB :
  - **FAISS** (local, simple), **pgvector** (Postgres), **Milvus** / **Qdrant** (production), **Weaviate**.
- Paramètres HNSW (ex.) : `M≈32`, `efConstruction≈200`, `efSearch≈64–128` (ajuste vs latence).

## 4.4 Retrieval & Re-ranking

- Retrieval top-k=20 (hybride: `alpha*BM25 + (1-alpha)*cosine`).
- Re-ranking top-5 via **cross-encoder** (bge-reranker-large, Cohere Rerank 3, etc.).
- Filtres: langue, collection, fraicheur (`last_updated >= now-180d`), permissions.

## 4.5 Contexte & Prompt

- Concatène **3–6 passages** max, ordonnés (titre→corps), *dedup*.
- Prompt: rôle, style, **exiger citations** `[[n]]` + “Réponds seulement à partir des sources; dis “je ne sais pas” sinon”.

## 4.6 Sécurité & conformité

- **PII redaction / access control par document**.
- Garde-fous sur sujets interdits (sécurité produit, légal).
- Journaux de consultation (qui a vu quoi).

## 4.7 Déploiement

- **API FastAPI** (Python) derrière Nginx.
- **Workers** ingestion (Celery/Cron).
- **Docker/Kubernetes**; variables secrets via Vault/SSM.
- Monitoring: **OpenTelemetry**, Prometheus, Grafana.

## 5) Stack d’outils (pratiques et crédibles)

- **LangGraph** (ou LangChain) pour les *pipelines/agents* et le contrôle de flux.
- **LlamaIndex** ou **Haystack** si tu préfères une librairie RAG “batteries-included”.
- **Vector Stores:** FAISS (PoC), **Qdrant/Milvus/pgvector** (prod).
- **Parsers:** Unstructured, Tika; **trafilatura** (HTML propre).
- **Rerankers**: Cohere Rerank, **bge-reranker-v2**, Voyage.
- **Eval: RAGAS**, **DeepEval**, **ARAGOG.**
- **Observabilité**: Langfuse, Phoenix, Helicone; tracing OpenTelemetry.
- **Guardrails: GuardrailsAI**, **NeMo Guardrails**, PII scrubbing (Presidio).
- **LLMs:** GPT-4.x / o3, Claude 3.x, Llama-3.1/3.2, Mistral-Large, Gemini; choisis selon data policy/latence/coût & multilingue.

## 6) Exemple minimal (Python, FAISS + rerank)

python

Copier le code

```python
# pip install faiss-cpu sentence-transformers rank_bm25 fastapi uvicorn
from sentence_transformers import SentenceTransformer, util
from rank_bm25 import BM25Okapi
import faiss, numpy as np

docs = load_docs()  # [{"id":..., "title":..., "text":..., "meta":{...}}, ...]
chunks = chunk_docs(docs)  # structure-aware splitting

embed = SentenceTransformer("mixedbread-ai/mxbai-embed-large-v1")
vecs = embed.encode([c["text"] for c in chunks], convert_to_numpy=True, normalize_embeddings=T

index = faiss.IndexHNSWFlat(vecs.shape[1], 32); index.hnsw.efConstruction = 200
index.add(vecs)

bm25 = BM25Okapi([c["text"].split() for c in chunks])

def retrieve(query, k=20, alpha=0.5):
    qv = embed.encode([query], normalize_embeddings=True)[0]
    D,I = index.search(np.array([qv]), k)  # ANN
    ann = [(i, float(sim)) for i, sim in zip(I[0], 1-D[0])]    # cosine approx

    bm = bm25.get_top_n(query.split(), list(range(len(chunks))), n=k)
    bm_scored = [(i, 1.0) for i in bm]

    # simple late fusion
    scores = {}
    for i,s in ann: scores[i] = scores.get(i,0)+ (1-alpha)*s
    for i,s in bm_scored: scores[i] = scores.get(i,0)+ alpha*s
```

## 7) Exemples d'applications

- **FAQ/Support** sur base de knowledge interne (politiques, procédures).
- **Copilot documentaire** (technique, juridique, médicale sous garde-fou).
- **Recherche code & design docs** (RAG sur dépôts Git + RFC internes).
- **RAG "données fraîches"** : feuilles de calcul, changelogs, tickets JIRA.
- **Tableaux de bord**: question → réponse sourcée + lien vers la page.

## 8) Évaluation de la qualité (indispensable en entretien)

- **RAGAS**: Answer Faithfulness, Context Precision/Recall, Context Relevance.
- **Human-in-the-loop**: gold Q/A + passages attendus.
- **A/B**: variantes de chunking (taille/overlap), top-k, reranker.
- **KPI prod**: taux de "je ne sais pas", taux de clic sur citations, temps de réponse, coût.

## 9) Performance & coût

- **Latence** : parallèle sur retrieval + re-ranking; **cache** embeddings & réponses.
- **Coût** : context ≤ 8–12k tokens; **summaries** des longs docs (index hiérarchique).
- **Scalabilité** : shards par collection/projet; `efSearch` dynamique selon SLA.

# 10) Sécurité / Gouvernance

- **RBAC par document** (filtre au niveau index).
- **Redaction PII** en ingestion.
- **Audit**: journaliser requêtes & sources retournées.
- **Isolation**: réseaux privés, chiffrage au repos & en transit.

## 11) Plan 30 / 60 / 90 jours (réaliste pour un employeur)

**30 jours (MVP)**

- Choisir stack (ex: FastAPI + LangGraph, Qdrant, bge-m3, Cohere Rerank).
- Ingestion pour 1–2 types de docs (PDF/HTML), chunking structuré.
- RAG hybride + rerank, citations, "je ne sais pas".
- RAGAS de base + tracing (Langfuse).

**60 jours (Pilotage)**

- RBAC, filtres métadonnées, collections multiples.
- Évaluation continue, dashboard des KPI.
- Agents légers: query rewriting, multi-hop sur cas difficiles.
- CI/CD + tests de régression qualité.

**90 jours (Prod élargie)**

- Observabilité complète, alerting coût/latence.
- Migrations d'index, rafraîchissements incrémentaux.
- Fine-tuning léger du reranker/domain adapters si besoin.
- Rollout équipes, feedback loop sur "flag bad answer".

## 12) Adapter à ton écosystème (Django / IDEO-Lab)

- **App "rag"**: modèles `Document`, `Chunk`, `Embedding`, `Collection`, `AccessControl`.
- **Management commands/Cron**: `scan_sources`, `index_docs`, `refresh_embeddings`.
- **Vector store**: `pgvector` si tu es déjà sur Postgres; sinon Qdrant (docker).
- **Vue Django**: une `qa_view(query, collection, user)` → JSON (answer + citations).
- **Admin**: statut d'ingestion, erreurs OCR, statistiques RAGAS.
- **Sécurité**: filtre par `user.groups` sur les requêtes de retrieval.

## 13) Questions fréquentes d'entretien (et pistes de réponse)

- **Comment réduire les hallucinations ?** RAG + prompt "répondre seulement avec sources" + "je ne sais pas" + eval.
- **Pourquoi un reranker ?** Les embeddings (bi-encodeurs) manquent de précision fine; cross-encoder optimise l'ordre final.
- **Chunk size optimal ?** Dépend du domaine; on teste 400–1 200 tokens + overlap; on évalue avec RAGAS.
- **Pourquoi hybride BM25+vecteur ?** Mots rares, symboles et "exact match" → BM25; sémantique → vecteur.
- **Gérer la fraîcheur ?** Métadonnées date, filtres, jobs d'index incrémental.
- **Et la sécurité ?** RBAC, PII redaction, audit, politiques de refus hors périmètre.

QUESTIONS REPONSES

## 1. Définition & Concept

Q: C'est quoi un RAG, et pourquoi est-ce utile ?

R: RAG = *Retrieval-Augmented Generation*. On enrichit un modèle de langage avec de l'information externe via un moteur de recherche sémantique. Ça réduit les hallucinations, permet de travailler avec des données fraîches, et donne des réponses sourcées.

## 2. Architecture générale

Q: Quels sont les composants principaux d'un pipeline RAG ?

R:

- **Ingestion & Parsing** : on extrait le texte des documents (PDF, HTML, DOCX).
- **Chunking** : on découpe les documents en morceaux adaptés.
- **Embeddings** : on projette chaque chunk en vecteur.
- **Index (Vector DB)** : pour rechercher efficacement les chunks pertinents.
- **Retrieval** : recherche top-k candidats (souvent hybride lexical + vecteur).
- **Re-ranking** : sélection fine avec un cross-encoder.
- **Prompt Assembly** : construction du contexte et injection dans le LLM.
- **Génération** : LLM produit la réponse, citations incluses.

## 3. Chunking

**Q:** Pourquoi découper les documents en chunks ? Quelle taille idéale ?

**R:**

- Parce qu'un LLM ne peut pas ingérer un document entier.
- Le chunking permet d'équilibrer **précision** (pas trop gros, sinon bruit) et **contexte complet** (pas trop petit, sinon perte de sens).
- Taille typique : 400–1 000 tokens, avec **overlap 10–20%** pour garder la continuité.

---

## 4. Embeddings

**Q:** Qu'est-ce qu'un embedding ?

**R:** C'est une représentation vectorielle dense du texte, dans un espace où la similarité cosinus reflète la proximité sémantique. Exemple : OpenAI `text-embedding-3-large`, BGE-m3, ou Voyage Multilingual.

**Q:** Comment choisir un modèle d'embedding ?

**R:**

- **Langue supportée** (multilingue vs monolingue).
- **Dimension & vitesse** (petit pour prod temps réel, large pour précision).
- **Licence & coût** (open source vs API managée).

## 5. Vector Databases

Q: Pourquoi utiliser une base vectorielle ?

R: Parce qu'on doit chercher les chunks les plus proches en **approximate nearest neighbors (ANN)** dans un espace à haute dimension. FAISS, Qdrant, Milvus, Weaviate, pgvector sont les plus utilisés.

---

## 6. Retrieval & Reranking

Q: Quelle différence entre retrieval et reranking ?

R:

- **Retrieval** = recherche rapide top-k candidats (ANN).
- **Reranking** = reclasser ces candidats avec un modèle plus coûteux mais précis (cross-encoder).
  Résultat : moins de bruit, meilleure pertinence.

Q: Pourquoi faire un retrieval hybride (lexical + vectoriel) ?

R: Parce que le vectoriel capture le sens, mais le lexical (BM25) est meilleur sur les mots rares, les acronymes ou les extraits de code. En combinant les deux → robustesse.

## 7. Prompt Engineering

Q: Comment construis-tu ton prompt dans un RAG ?

R:

- **Instructions** ("Réponds uniquement à partir des sources suivantes").
- **Contexte** (les passages retrouvés).
- **Format attendu** (citations `[[n]]`, style concis, JSON si structuré).
- **Fallback** : "Si tu ne sais pas, dis-le".

---

## 8. Hallucinations & Qualité

Q: Comment réduire les hallucinations dans un RAG ?

R:

1. Retrieval de qualité (hybride + rerank).
2. Prompt clair ("Réponds seulement si contexte pertinent").
3. Autoriser le modèle à dire "je ne sais pas".
4. Évaluation continue (RAGAS, human review).

## 9. Évaluation

Q: Comment évaluer la qualité d'un RAG ?

R: Avec des métriques comme :

- **Faithfulness** (la réponse s'appuie bien sur les sources).
- **Context Recall** (les bons passages ont été retrouvés).
- **Context Precision** (pas trop de bruit).
- **Réponse correcte vs gold data.**

  Outils : **RAGAS**, **DeepEval**, ou jeux de tests internes.

---

## 10. Scalabilité & Perf

Q: Comment scaler un RAG pour des millions de documents ?

R:

- Indexation incrémentale (ne pas recalculer tout).
- Sharding de la base vectorielle.
- Cache des embeddings & réponses.
- Ajuster `efSearch` ou `top-k` selon SLA.

## 11. Sécurité & Gouvernance

Q: Comment intégrer de la sécurité dans un RAG ?

R:

- **RBAC** : filtrage des documents par utilisateur/groupe.
- **Redaction** : suppression des PII avant ingestion.
- **Audit** : journalisation des requêtes & sources retournées.
- **Guardrails** : filtrer sujets interdits, gestion du refus.

---

## 12. Exemples concrets

Q: Donne-moi un cas d'usage concret d'un RAG en entreprise.

R:

- **Support client** : chatbot qui répond à partir des guides produits et tickets historiques.
- **Recherche juridique** : avocat qui interroge sa base de jurisprudence interne.
- **Recherche technique** : ingénieur qui pose des questions sur le code interne + docs API.

## 13. Mise en prod

Q: Comment déploies-tu un RAG ?

R:

- Backend **FastAPI** ou Django pour exposer `/ask`.
- Base vectorielle (pgvector, Qdrant).
- Workers ingestion (Celery, Cron).
- Monitoring (Prometheus, Grafana, Langfuse).
- Déploiement containerisé (Docker, Kubernetes).

---

## 14. Pièges & bonnes pratiques

Q: Quels sont les principaux pièges dans un projet RAG ?

R:

- **Chunking mal fait** → retrieval inutilisable.
- **Pas de rerank** → réponses bruitées.
- **Index pas mis à jour** → données obsolètes.
- **Pas d'évaluation continue** → dérive non détectée.

QUESTIONS ENTRAINEMENT

## 1. Définition

**Q:** C'est quoi un RAG ?

**R:** Retrieval-Augmented Generation : une architecture qui combine **recherche sémantique + LLM génératif** pour donner des réponses basées sur des documents pertinents.

---

## 2. Objectif

**Q:** Pourquoi utiliser un RAG plutôt que du fine-tuning ?

**R:** Parce que le RAG est **plus flexible**, **moins coûteux**, met à jour les connaissances sans réentraîner le modèle, et réduit les hallucinations.

---

## 3. Pipeline

**Q:** Quelles sont les grandes étapes d'un pipeline RAG ?

**R:** Ingestion → Parsing → Chunking → Embeddings → Index vectoriel → Retrieval → Re-ranking → Prompt → LLM → Réponse sourcée.

---

## 4. Chunking

**Q:** Pourquoi découper les documents en chunks ?

**R:** Pour que chaque morceau reste compréhensible par le modèle, tout en évitant de dépasser la taille de contexte. Taille typique : 400–1 000 tokens.

## 5. Embeddings

**Q:** À quoi servent les embeddings ?

**R:** À représenter le texte sous forme de vecteurs dans un espace où la **similarité** reflète la **proximité sémantique**.

---

## 6. Vector DB

**Q:** Pourquoi utiliser une base vectorielle ?

**R:** Pour retrouver rapidement les passages pertinents via **approximate nearest neighbors (ANN)**. Ex : FAISS, Qdrant, Milvus, Weaviate.

---

## 7. Hybrid Retrieval

**Q:** C'est quoi le retrieval hybride ?

**R:** C'est la combinaison du **lexical** (BM25) et du **vectoriel** pour améliorer la robustesse, surtout sur les mots rares ou le code.

---

## 8. Re-ranking

**Q:** Pourquoi ajouter un reranker ?

**R:** Parce que le retrieval brut peut donner du bruit. Le **reranker** (cross-encoder) améliore la pertinence finale.

## 9. Prompt

**Q:** Comment construire un prompt efficace dans un RAG ?

**R:** Inclure les passages sources, donner une instruction claire (“Réponds uniquement à partir de ces sources”), et prévoir un fallback (“Je ne sais pas”).

---

## 10. Hallucinations

**Q:** Comment limiter les hallucinations ?

**R:** Avec un retrieval de qualité, un prompt restrictif, un “je ne sais pas”, et de l’évaluation continue.

---

## 11. Évaluation

**Q:** Comment évaluer un système RAG ?

**R:** Avec des métriques comme **faithfulness**, **context recall/precision**, et des outils comme **RAGAS** ou **DeepEval**.

---

## 12. Scalabilité

**Q:** Comment gérer un corpus de plusieurs millions de documents ?

**R:** Indexation incrémentale, sharding du vector store, cache d’embeddings, et paramètres ANN ajustés.

## 13. Mise à jour

**Q:** Comment maintenir un RAG à jour ?

**R:** Avec une ingestion incrémentale, une base versionnée, et un rafraîchissement régulier des embeddings.

---

## 14. Langues

**Q:** Comment gérer le multilingue dans un RAG ?

**R:** Utiliser des embeddings multilingues (ex. BGE-m3, LaBSE), et stocker la langue en métadonnée pour filtrer.

---

## 15. Sécurité

**Q:** Comment contrôler l'accès aux données dans un RAG ?

**R:** Avec du **RBAC** (filtrage par utilisateur/groupe) au niveau retrieval, et une gestion fine des permissions.

---

## 16. Confidentialité

**Q:** Comment protéger les données sensibles ?

**R:** Chiffrement, redaction des PII avant ingestion, et audit des accès.

## 17. Performances

**Q:** Comment réduire la latence d'un RAG ?

**R:** Caching des embeddings/réponses, parallélisation retrieval + rerank, ajustement du top-k.

---

## 18. LLM

**Q:** Quel modèle de langage choisir pour un RAG ?

**R:** Ça dépend : GPT-4o pour qualité, Claude/Mistral pour coûts et gouvernance, Llama-3 pour self-hosting.

---

## 19. Exemple concret

**Q:** Donne un cas d'usage typique d'un RAG.

**R:** Un chatbot support client qui répond à partir de la documentation produit interne et fournit les sources.

---

## 20. Fallback

**Q:** Que faire si aucun passage pertinent n'est trouvé ?

**R:** Répondre "Je ne sais pas" ou proposer de reformuler la question, plutôt que générer une réponse inventée.

## 21. Chunk overlap

**Q:** Pourquoi ajouter de l'overlap entre les chunks ?

**R:** Pour préserver la continuité sémantique entre deux morceaux coupés.

---

## 22. Mauvais chunking

**Q:** Que se passe-t-il si le chunking est trop petit ou trop grand ?

**R:** Trop petit = perte de contexte, trop grand = bruit et dépassement du contexte LLM.

---

## 23. Outils ingestion

**Q:** Quels outils pour parser les documents ?

**R:** Unstructured, Apache Tika, trafilatura, Tesseract pour OCR.

---

## 24. Caching

**Q:** Pourquoi mettre en place un cache ?

**R:** Pour réutiliser les embeddings déjà calculés, éviter des appels coûteux et réduire la latence.

## 25. Monitoring

**Q:** Comment monitorer un RAG en prod ?

**R:** Avec des outils comme **Langfuse, Phoenix, Helicone** + traces OpenTelemetry, métriques (latence, coût, qualité).

---

## 26. RAG vs Fine-tuning

**Q:** Quelle différence entre RAG et fine-tuning ?

**R:** Fine-tuning modifie les poids du modèle → coûteux, figé. RAG ajoute une couche de retrieval → flexible, mis à jour facilement.

---

## 27. Guardrails

**Q:** C'est quoi les guardrails dans un RAG ?

**R:** Des règles de sécurité qui contrôlent ce que le LLM peut dire, filtrent les inputs/outputs, et gèrent le refus hors périmètre.

---

## 28. Écosystème

**Q:** Quelles librairies utiliser pour implémenter un RAG ?

**R:** LangChain/LangGraph, LlamaIndex, Haystack, FAISS, Qdrant, Milvus, pgvector.

## 29. Équipe

**Q:** Qui travaille sur un projet RAG en entreprise ?

**R:** Ingénieurs IA, Data Engineers (pipeline ingestion), DevOps (déploiement/monitoring), Product Owners (use cases).

---

## 30. Limites

**Q:** Quelles sont les limites d'un RAG ?

**R:** Dépendance à la qualité des données, coût en tokens si contexte trop large, latence si retrieval mal optimisé.

---

# PLAN D’APPROCHE COMPLET

## 1) Cadrage (Jours 1–3)

- **Objectifs métier** : types de questions, niveau de précision attendu, langues, latence max, coûts/req, périmètre des sources.
- **KPI & critères d'acceptation**
  - *Retrieval*: hit@k, MRR, nDCG.
  - *QA*: Exact-Match / F1, Faithfulness (hallucination≤X%), Latence P50/P95, Coût/req.
- **Conformité** : GDPR (base légale, minimisation), PII (masquage), rétention & traçabilité.
- **Livrables** : doc de cadrage + backlog EPICs/US + protocole d'évaluation initial.

## 2) Cartographie & ingestion des données (Jours 3–10)

- **Sources** : HTML/MD, PDF, DOCX, Confluence, Notion, DB, S3/GCS, Git.
- **Normalisation** : extraire texte + métadonnées (titre, auteur, date, URL, chemin), nettoyer (tableaux, notes, headers/footers).
- **Versioning & fraîcheur** : champs `source_id`, `version`, `hash`, `last_modified`, flags `is_current`.
- **Livrables** : connecteurs d'ingestion + schéma des métadonnées + pipeline incrémental (Cron/Celery).

## 3) Stratégie de chunking & enrichissement (Jours 7–12)

- **Chunking hiérarchique** :
  - *Par structure* (H1/H2/H3, sections),
  - *Fenêtre glissante* (overlap 15–25%),
  - *Taille* 400–800 tokens selon domaine.
- **Contexte augmenté** :
  - Mots-clés (RAKE/KeyBERT), ancrages (titres parents), embeddings de *section headers*, tables rendues en texte.
- **Qualité** : détection de duplicats, score de lisibilité, longueur utile.
- **Livrables** : config YAML de chunking + rapport d'ablation (taille/overlap → hit@k).

## 4) Indexation & vecteurs (Jours 10–15)

- **Embeddings** : base multilingue (ex. bge-m3 / e5-large / text-embedding-3-large), fixer `dim` et norm L2.
- **Index** :
  - PoC rapide → **FAISS** (Flat/HNSW)
  - Prod → **pgvector** (si SQL first) ou **Elasticsearch/OpenSearch + dense vectors**, ou **Weaviate/Milvus** si volumétrie élevée.
- **Filtres** : métadonnées (langue, business unit, fraicheur<90j, type=documentation/policy).
- **Livrables** : index créé + scripts de (re)build + tests de rappel (recall).

## 5) Orchestration RAG (Jours 12–18)

- **Schéma canonique de requête**
  1. *Reformulation* (optional): nettoyer la question, détecter langue & intent.
  2. *Retrieval* : hybrides (BM25 + vecteur), k=20 puis *reranking* (Cross-Encoder) → top p=5.
  3. *Construit le prompt* : instructions + `context windows` (citations & métadonnées).
  4. *Génération* : modèle ciblé (latence/coût), *constrained style* (citations, ton neutre).
  5. *Post-process* : vérif factualité (self-check/consistency), ajout des sources.
- **Guardrails** : refus hors périmètre, PII leak, jailbreak, classification sécurité.
- **Livrables** : service `/rag/ask` (REST/GraphQL), schémas pydantic, logs structurés.

## 6) Prompts & gabarits (Jours 15–20)

- **System prompt** (extraits)
  - « Tu réponds **uniquement** avec les passages fournis ; si insuffisant → "Je n'ai pas l'info" »
  - « Résume en N points, cite les sources `[source:title](url)` »
- **Templates** : `qa_basic`, `qa_step_by_step`, `qa_tabular`, `qa_code`, `qa_policy`.
- **Langues** : prompt bilingue, réponses dans la langue de la question.
- **Livrables** : bibliothèque de prompts + tests unitaires (snapshot).

## 7) Évaluation offline & bancs d'essai (Jours 18–24)

- **Jeux d'éval** : 100–300 Q/A dorées + *hard negatives* + multi-versions (v1…vn).
- **Metrics** : EM/F1, Faithfulness (LLM-as-a-judge + contrôle n-grammes), hit@k, MRR, nDCG, latence P50/P95, coût/req.
- **Ablations** : embeddings A/B, chunk size, reranker ON/OFF, hybrid vs dense-only.
- **Livrables** : tableau de bord d'éval + rapport de choix techniques.

## 8) Observabilité & qualité en run (Jours 22–28)

- **Tracing** : par étape (reformulation, retrieve, rerank, answer).
- **Feedback boucle-fermée** : like/dislike, correction humaine → ré-indexation priorisée.
- **Alertes** : hausse latence, chute hit@k, coût anormal.
- **Red teaming continu** : jeux d'attaques (prompt-injection, data exfiltration).
- **Livrables** : graphiques & alertes (Prometheus/Grafana + logs JSON).

## 9) Sécurité, conformité, gouvernance (en parallèle)

- **PII** : masquage à l'ingestion + filtres de restitution.
- **GDPR** : registre des traitements, DPA fournisseurs, droit à l'oubli (purge ciblée).
- **RBAC** : accès aux sources par rôle, filtrage au *retrieval* (ex. `business_unit in user.claims`).
- **Livrables** : matrice d'accès + procédures purge & rotation clés.

## 10) Déploiement & coûts (Jours 24–30)

- **Environnements** : Dev/Staging/Prod, variables sécurisées, clés KMS.
- **Infra** : API Python (FastAPI ou Django view), workers Celery, base documents + vecteurs, cache Redis.
- **Performance** : batch embeddings, warm pools, max tokens & truncation, streaming réponses.
- **Budget** : coût/1k requêtes (embeddings + génération + stockage), objectif < X €.
- **Livrables** : IaC (Terraform/Ansible), runbook, matrices de scaling.

---

### Architecture de référence (simple & robuste)

- **Ingestion**: Python workers → stockage *documents_raw* (S3/FS) + *documents_clean* (SQL).
- **Index**: pgvector/FAISS + index BM25 (Postgres/ES).
- **Rerank**: Cross-Encoder (petit modèle local pour coûts).
- **API**: `/rag/ask` + `/rag/eval` + `/rag/admin` (reindex, purge).
- **Auth**: JWT / session Django ; RBAC en filtre de retrieval.
- **Observabilité**: logs JSON, traces, métriques.

## Modèle de données (SQL)

**documents**(id, title, url, lang, doc_type, source_id, version, last_modified, hash, is_current)

**chunks**(id, document_id, chunk_no, text, tokens, headers_path, lang, embedding[dim], created_at)

**indexes**(type, params_json, created_at)

**qa_feedback**(id, user_id, question, answer, sources_json, rating, comment, created_at)

**events**(ts, stage, latency_ms, cost_usd, meta_json)

## API contract (exemple)

POST `/rag/ask`

Body:

json Copier le code

```
{
  "question":"Comment migrer de MySQL vers Postgres ?",
  "lang":"fr",
  "filters":{"doc_type":["guide","runbook"],"fresh_days":180},
  "top_k":5,
  "stream":true
}
```

Response:

json Copier le code

```
{
  "answer":"…",
  "citations":[{"title":"Runbook migration","url":"...","chunk_id":123}],
  "latency_ms":842,
  "cost_usd":0.0023
}
```

## Prompt gabarit (extrait)

makefile

Copier le code

```
SYSTEM: Tu es un assistant factuel. Réponds uniquement à partir des extraits f
Si l'information manque, dis-le franchement et propose la source la plus proch
Inclue une section "Sources" avec [Titre](URL). Langue = {user_lang}.

USER: {question}

CONTEXT (max 5 extraits, ≤1500 tokens):
1) {chunk_1_text}  (meta: {title_1} • {url_1})
...
```

## Mini plan de tests

- **Unitaires** : parsers PDF/HTML, chunking, embeddings (dim cohérentes), filtre RBAC.
- **Intégration** : `/rag/ask` end-to-end, erreurs réseau, timeouts LLM.
- **Charge** : 100–500 RPS en burst, profiling P95.
- **Sécurité** : prompt-injection, SSRF via URLs de sources, exfiltration PII.

---

## Roadmap indicative (4 semaines)

- **S1** : cadrage, ingestion v1, chunking, index vecteur BM25+FAISS.
- **S2** : API `/rag/ask`, prompts, reranking, éval offline v1.
- **S3** : observabilité, guardrails, RBAC, feedback loop.
- **S4** : hardening, coûts/latence, doc, déploiement prod.

---

## Risques & parades

- **Hallucinations** → prompts stricts + self-check + citations obligatoires.
- **Qualité retrieval** → hybrid + rerank + tuning chunking.
- **Coûts** → embeddings batch + cache + modèles compacts.
- **Données sensibles** → masquage, RBAC au retrieval, audits.
- **Fraîcheur** → re-ingestion incrémentale + `last_modified` watcher.

## Livrables finaux

1. Infra RAG déployée (Dev/Prod) + IaC.
2. Connecteurs & pipeline ingestion/refresh.
3. Index hybride + scripts de maintenance.
4. API publique + doc OpenAPI + exemples.
5. Tableau de bord d'évaluation + rapports d'ablation.
6. Observabilité (metrics/traces/logs) + alertes.
7. Guide d'exploitation & sécurité (RBAC, GDPR, purge).

# APPLICATION CONCRETE DJANGO DATAMODEL

## 0) Cible produit (ce que le RAG doit savoir faire)

- **Questions de design** : "Faut-il `ManyToMany` ou `ForeignKey` ?", "Comment modéliser un soft-delete propre ?"
- **Génération guidée** : squelettes de `models.py`, `Meta`, `constraints`, `index_together`, `unique_together`.
- **Migrations** : proposer `makemigrations` attendues (opérations `AddField`, `AlterField`, `RunSQL`, `RunPython`).
- **Compatibilité** : vérifier l'impact sur données, perfs, admin, DRF serializers, forms.
- **Conformité** : conventions de ton projet (naming, `BigAutoField`, `db_table`, `on_delete`, `related_name`).

---

## 1) Sources & périmètre (ce que tu ingères)

1. **Code Django**
   - `apps/**/models.py`, `apps/**/apps.py`, `apps/**/admin.py`, `apps/**/serializers.py`, `apps/**/signals.py`
   - `settings.py`, `DATABASES`, `AUTH_USER_MODEL`, `INSTALLED_APPS`
   - Migrations: `apps/**/migrations/*.py` (historique + opérations)
2. **Base de données**
   - Schéma réel (via introspection): tables, colonnes, index, FK, contraintes, vues matérialisées
   - Volumétrie & cardinalités (échantillon anonymisé)
3. **Docs & conventions internes**
   - READMEs, "coding guidelines", glossaire métier, règles de nommage
4. **Traçabilité**
   - `graph_models` (django-extensions) → diagrammes, ou export ERD (pydot)

## 2) Ingestion & parsing (structure-aware)

- **Python/AST parsing** (ou Tree-sitter) pour extraire:
  - `class Model(models.Model)` : nom, app_label, docstring
  - Champs: type ( `CharField` , `JSONField` , …), args ( `null` , `blank` , `db_index` , `choices` , `validators` )
  - `class Meta` : `db_table` , `indexes` , `constraints` , `ordering` , `unique_together`
- **Migrations parser**
  - Liste chronologique des opérations ( `AddField` , `AlterUniqueTogether` , `RunSQL` , …)
  - Déduire l'**état courant** du modèle (après relecture séquentielle)
- **DB introspection**
  - `SHOW CREATE TABLE` (MySQL/MariaDB) / `information_schema` / `pg_catalog` (Postgres)
  - Capturer index manquants, FKs orphelines vs models
- **Admin/DRF**
  - `ModelAdmin` (list_filter, search_fields) ; `ModelSerializer` (fields, validators)

> Sortie ingestion = **objets structurés** (JSON) + **chunks textuels** riches en contexte.

## 3) Chunking (pensé “DataModel”)

- **Granularité**:
  - 1 chunk **par modèle** (entête + champs + Meta + docstring)
  - 1 chunk **par migration** significative (groupées par feature)
  - 1 chunk **par contrainte/index** (avec justification)
  - 1 chunk **par convention projet** (naming, on_delete, timezone, IDs)
- **Contexte adjoint**:
  - Joindre **imports utiles** (abstract base, mixins)
  - Pour migrations : inclure **avant/après** synthétique
- **Taille**: 400–900 tokens, **overlap** 80–150 tokens entre sections voisines

---

## 4) Index & embeddings

- **Vector store**:
  - **pgvector** si tu es déjà sur Postgres ; sinon **Qdrant** (docker) pour prod.
- **Embeddings** (texte & code):
  - Multilingue & code-friendly (ex. BGE-m3 / jina-embeddings-code / e5-base-v2)
- **Lexical index** (hybride):
  - BM25 (Whoosh/Elastic/OpenSearch) pour mots clés, noms de tables/champs, symboles
- **HNSW params** (ex. Qdrant): `m≈32`, `ef_construct≈128–256`, `ef_search≈64–128`

## 5) Retrieval & re-ranking

- **Hybrid late fusion**: score = `α*BM25 + (1-α)*cosine` ; `α` typique 0.35–0.55
- **Filtres:** `app_label`, `model_name`, `kind` (model/migration/constraint), `version`, `lang`
- **Re-ranking**:
  - Cross-encoder (ex. `bge-reranker-base` / Cohere Rerank) sur top-k=20 → top-5 final
- **Spé DataModel**:
  - Boost si le chunk **mentionne** la table/champ de la question
  - Boost si `last_migration_date` récente (évolution active)

---

## 6) Assemblage du contexte (Context Builder)

- **Merge intelligent**:
  - Regrouper *modèle + contraintes + admin + serializer* pour une vue 360°
  - Dédupliquer, trier : *définition → contraintes → migrations → usages (admin/DRF)*
- **Budget token:** viser 3–6 passages (≤ 8–12k tokens total prompt)
- **Snippets ciblés**: inclure **signature de champ** complète et `Meta`

# 7) Prompts spécialisés “DataModel”

## 7.1 Design / refactor de modèle

> Rôle: Tu es un architecte Django. En t’appuyant **uniquement** sur les sources ci-dessous, propose une conception de modèle.
>
> Contrainte: respecter conventions du projet (voir section “Conventions”).
> **Livrables**:
>
> 1. “**Diff** `models.py` (code)”
> 2. “**Opérations de migration** attendues”
> 3. “**Impacts** (données, index, performances)”
> 4. “**Risques** & mitigations”
> 5. “**Citations** des sources `[[n]]`”

## 7.2 Aide migration

> Entrée: “Ajouter `unique_together` sur (user, project) pour `Membership` + index sur `status`.”
>
> Sortie: snippet `migrations/000X_auto.py` avec `AlterUniqueTogether`, `AddIndex`, et script `RunPython` si data-fix nécessaire.

## 7.3 Audit de cohérence

> Vérifier écarts **models vs DB** (types, nullability, longueurs), suggérer corrections.

## 8) Sécurité, confidentialité, gouvernance

- **Filtrage** par repo/projet/équipe : RBAC appliqué au retrieval
- **Redaction:** purge secrets ( `SECRET_KEY` , DSN, mots de passe, clés API)
- **Isolation:** index privés par environnement (dev/stage/prod)
- **Traçabilité:** logs des questions + passages fournis + coût/latence

---

## 9) Observabilité & évaluation (qualité)

- **Telemetry:** Langfuse/OpenTelemetry (latence, token, hits cache)
- **Qualité:**
  - **RAGAS:** faithfulness, context relevance, answer correctness
  - **Tests de régression:** 50–100 Q/A internes (voir §11)
- **KPIs:** % réponses "actionnables", % "je ne sais pas", temps de réponse, incidents migrations évités

## 10) Backend & composants (référence)

- **API**: FastAPI (ou Django Rest Framework) → `POST /rag/datamodel/ask`
- **Workers ingestion**: Celery/cron
- **Jobs**:
  - `scan_repo` (git clone/pull, diff)
  - `parse_models`, `parse_migrations`, `introspect_db`
  - `index_chunks` (embeddings + BM25)
- **Stockage RAG (Django app `rag_dm`)**
  - `Document(id, path, app_label, kind, version, ts, meta)`
  - `Chunk(id, document_id, text, tokens, model_name, meta)`
  - `Embedding(id, chunk_id, vector, model)`
  - `QueryLog(id, user, question, answer, cost, latency, citations[])`
  - `EvalSample(id, question, expected_points[], golden_chunks[])`

---

## 11) Jeu d'évaluation (exemples concrets)

- "Dans `orders`, comment modéliser un `OrderLine` avec TVA variable et remise ?"
- "Faut-il `ManyToMany(through)` pour `User` ↔ `Team` si on ajoute un rôle, une date d'entrée, un statut ?"
- "Ajouter une **contrainte** : `(start_date <= end_date)` sur `Booking`."
- "Mettre en place **soft-delete** (champ `deleted_at`) et filtrage par `QuerySet` manager."
- "Comment migrer `CharField(choices)` → `PositiveSmallIntegerField` + `TextChoices` sans casser la prod ?"
- "Indexer `created_at` & `(status, created_at)` : impact perfs/écritures ?"
- "Multi-tenant : schéma partagé, champ `tenant_id` + `unique_together(tenant_id, …)`."
- "Audit des **nullability** incohérentes entre DB et models."

## 12) Expérience utilisateur (UI/UX interne)

- **Playground:** champ question + sélecteur `app_label` + scope (models/migrations/constraints)
- **Affichage réponse**:
  - **Onglets**: *Proposition code* | *Migrations* | *Impacts* | *Sources*
  - Boutons: “Copier code”, “Créer PR”, “Exporter patch”
- **Garde-fous:** bannière “Vérifiez avant merge — génère un diff Git”

## 13) 30 / 60 / 90 jours (feuille de route réaliste)

**J+30 (MVP)**

- Ingestion `models.py` + migrations + conventions
- Index **hybride** (pgvector + BM25)
- Endpoint `/ask` + prompt “Design de modèle”
- 20 Q/A RAGAS + tracing

**J+60 (Pilotage)**

- Introspection DB + audit cohérence models↔DB
- Re-ranking cross-encoder
- Génération **migrations** + “impact analysis”
- RBAC par repo/projet ; dashboard qualités

**J+90 (Prod)**

- Collections multi-projets, multi-langues
- Agents: **query rewriting** (*“je cherche un identifiant court” → “slug/ShortUUID”*)
- PR auto-draft (GitHub/GitLab API) + tests de régression qualité à chaque indexation
- Alertes coût/latence, A/B chunking