# Assistant IA low-cost pour la Nuit de l'Info 2025

Architecture technique et justification des choix IA

Équipe Nuit de l'Info 2025 – FST Nouakchott

5 décembre 2025

## 1 Contexte et objectifs

La **Nuit de l'Info** est une compétition nationale de développement web où des équipes d'étudiants conçoivent, en une nuit, une application autour d'un sujet donné. Dans ce cadre, nous avons développé un **assistant IA low-cost** pour répondre aux questions fréquentes sur l'événement (organisation, inscription, défis, technique, etc.).

Les objectifs principaux sont :

— offrir un **assistant bilingue** (français / arabe) utilisable pendant la nuit ;
— rester **robuste en contexte de faible connectivité** : fonctionnement même avec une connexion lente ou absente ;
— limiter les **coûts d'infrastructure** (pas de GPU, pas de modèle local lourd) ;
— proposer **trois modes complémentaires** : Offline, Hybride et Online, avec bascule automatique ou manuelle.

## 2 Vue d'ensemble de l'architecture

Le projet est organisé en trois grands blocs :

1. **Frontend** (`frontend/`) : application React (PWA) qui gère l'interface de chat, la détection de la qualité de connexion, le stockage local (IndexedDB) et la sélection des modes IA.
2. **Backend** (`backend/`) : API FastAPI exposant `/api/chat` et `/api/health` pour le mode Online, avec un module de RAG (Retrieval-Augmented Generation) et un client LLM connecté à OpenRouter.
3. **Pipeline de données** (`Script/` + `process_all_data.py`) : scraper du site `nuitdelinfo.com`, génération d'une base de FAQs et d'embeddings 384D low-cost, sauvegardés pour le frontend.

D'un point de vue déploiement :

— le **frontend** peut être servi comme site statique (Vercel, Netlify, GitHub Pages, etc.) ;
— le **backend** tourne sur un petit serveur ou conteneur (FastAPI + Uvicorn, sans besoin de GPU) ;
— le pipeline de données est lancé ponctuellement pour mettre à jour les FAQ et les embeddings.

## 3 Architecture détaillée

### 3.1 Structure du dépôt

La racine du projet contient notamment :

— `frontend/` : application React (PWA, chat, modes IA, i18n) ;

— `backend/` : API FastAPI avec RAG + appel LLM ;
— `Script/` : scraper du site Nuit de l'Info et données brutes ;
— `process_all_data.py` : génération des FAQs et embeddings ;
— `frontend/public/data/faqs.json` : base de FAQs bilingues ;
— `frontend/public/data/embeddings.json` : embeddings 384D low-cost.

### 3.2 Frontend : PWA, chat et modes IA

Le frontend (`frontend/`) est une **Progressive Web App** qui fonctionne dans le navigateur :

— **PWA** : manifest, Service Worker, installation sur écran d'accueil mobile ou desktop, cache pour une utilisation offline partielle.
— **i18n** : interface disponible en français et arabe, avec gestion du sens de lecture RTL (right-to-left) pour l'arabe.
— **Stockage local** : `IndexedDB` via un gestionnaire dédié pour stocker les FAQs, les embeddings et éventuellement l'historique minimal des conversations.
— **Hooks et services IA** :
  — `useChat` : gestion de l'état du chat, envoi des questions, réception des réponses, affichage des sources ;
  — `useConnection` : estimation de la qualité réseau (bon, moyen, mauvais) ;
  — `AIService` : point d'entrée qui sélectionne quel moteur IA utiliser (Offline, Hybride, Online) en fonction de la connexion et d'un éventuel mode manuel.

### 3.3 Backend : FastAPI, RAG et client LLM

Le backend (`backend/app/`) est une API FastAPI minimaliste :

— `main.py` : crée l'application, configure CORS, et expose :
  — `/api/health` : renvoie un statut simple (ok / error) ;
  — `/api/chat` : reçoit une question et renvoie une réponse enrichie par le LLM.
— `rag.py` : charge `faqs.json`, recalcule des embeddings compatibles avec ceux du frontend, et applique une recherche sémantique (similarité cosinus) pour retrouver les FAQs les plus proches de la question.
— `llm_client.py` : construit un **prompt** en utilisant les FAQs les plus pertinentes (contexte RAG) et appelle l'API OpenRouter avec un modèle de type : `meta-llama/llama-3.3-70b-instruct:free`
— `config.py` : centralise la configuration (chemins vers les données, variables d'environnement `OPENROUTER_API_KEY` et `OPENROUTER_MODEL`, etc.).

Le backend ne connaît pas toute la base de données brute ; il se base sur les fichiers déjà préparés pour le frontend (`faqs.json`) et applique une logique de **RAG** avant l'appel LLM.

### 3.4 Pipeline de données : scraping et FAQ intelligente

Le scraping du site officiel est réalisé par le script `Script/scraper_nuit_info.py` :

— démarrage depuis quelques URLs de base sur `nuitdelinfo.com` ;
— exploration en largeur (BFS) des liens internes HTML ;
— filtrage pour éviter les sections « lourdes » (par exemple, certaines pages d'équipes ou d'archives massives) ;
— limite sur le nombre total de pages et délai minimal entre deux requêtes pour **éviter les crawls infinis** ;
— récupération de différents types d'informations (textes, liens, images, PDFs, vidéos, boutons, formulaires) dans des fichiers JSON (par exemple `texts.json`, `links.json`, etc.) dans `Script/data/nuit_info/`.

Le script `process_all_data.py` :

1. charge les données scrappées ;
2. construit une **liste manuelle mais structurée de FAQs** bilingues (FR/AR), organisée par catégories (général, organisation, inscription, défis, technique, évaluation, assistant, modes) ;
3. ajoute éventuellement une FAQ spécifique sur les vidéos si des vidéos ont été détectées pendant le scraping ;
4. enregistre les FAQs dans `frontend/public/data/faqs.json` avec des métadonnées (version, date de mise à jour, catégories, etc.) ;
5. génère des **embeddings 384D low-cost** pour chaque FAQ et les sauvegarde dans `frontend/public/data`

# 4 Embeddings low-cost : approche déterministe 384D

Pour rester dans une logique « low-cost », nous n'utilisons pas de modèle pré-entraîné lourd (type BERT, sentence-transformers, etc.) afin d'éviter le besoin de GPU et de gros téléchargements.

À la place, nous utilisons une technique **hash-based** simple et déterministe :

— texte d'entrée : concaténation « question_fr + réponse_fr » ;
— pré-traitement : minuscule, découpage en mots ;
— initialisation d'un vecteur $v \in R^{384}$ à zéro ;
— pour chaque mot $w$, calcul d'un hash $h(w)$, puis $i = h(w) \bmod 384$ et $v_i \leftarrow v_i + 1$ ;
— normalisation finale du vecteur $v$ (norme euclidienne).

Cette méthode :

— est **très rapide** (quelques opérations par mot) ;
— ne requiert **aucun modèle externe** ni GPU ;
— est **reproductible** entre Python (backend) et JavaScript (frontend), ce qui permet d'utiliser exactement la même logique d'embedding dans les deux mondes ;
— fournit des vecteurs de dimension 384 (dimension inspirée de MiniLM), suffisante pour distinguer quelques dizaines de FAQs.

Les embeddings sont exploités dans :

— le **mode Hybride** côté frontend, pour faire une recherche sémantique locale ;
— le backend (module `rag.py`) pour retrouver les FAQs les plus proches avant l'appel au LLM.

# 5 Les trois modes IA

## 5.1 Mode Offline

Le mode **Offline** est purement local :

— les FAQs sont stockées dans IndexedDB ;
— la recherche se fait par **mots-clés** (matching simple) ;
— aucune requête réseau n'est nécessaire ;
— temps de réponse typique : $< 200$ ms.

Ce mode est utilisé lorsque la connexion est coupée ou très instable. Il couvre les questions les plus fréquentes avec des réponses directes.

### 5.2 Mode Hybride

Le mode **Hybride** repose sur les embeddings 384D :

— le frontend calcule l'embedding de la question utilisateur avec la même fonction hash-based que le backend ;
— il compare ce vecteur aux embeddings des FAQs (`embeddings.json`) via la **similarité cosinus** ;
— il sélectionne la FAQ la plus proche (ou les $k$ plus proches) et renvoie la réponse correspondante.

Ce mode a plusieurs avantages :

— tout se passe **localement** (pas de LLM, pas de requête réseau) ;
— il gère mieux les reformulations, synonymes et fautes de frappe que la simple recherche par mots-clés ;
— il reste utilisable même avec un débit faible.

### 5.3 Mode Online

Le mode **Online** est activé lorsque la connexion est suffisamment bonne. Il introduit un **LLM distant** via OpenRouter :

1. le frontend envoie la question au backend FastAPI via `/api/chat` ;
2. le backend recharge les FAQs et calcule l'embedding de la question ;
3. un module RAG sélectionne quelques FAQs les plus pertinentes ;
4. ces FAQs sont injectées dans un **prompt** structuré, avec des consignes claires pour que le modèle :
   — reformule la réponse dans ses propres mots ;
   — indique les sources (titres ou identifiants de FAQ) ;
   — ne se contente pas de copier-coller la FAQ brute.
5. le backend renvoie au frontend une réponse en langage naturel, augmentée d'informations de contexte (sources, score de confiance, mode utilisé).

En cas de timeout, d'erreur HTTP ou de problème de clé API, le frontend retombe automatiquement sur le mode Hybride pour conserver une **expérience dégradée mais fonctionnelle**.

## 6 Justification des choix IA low-cost

### 6.1 Contraintes et enjeux

Les principaux enjeux qui orientent l'architecture sont :

— **Accessibilité** : permettre à des établissements avec peu de moyens (machines anciennes, connexion limitée) d'utiliser l'assistant ;
— **Simplicité de déploiement** : pouvoir héberger le frontend sur un simple service statique et le backend sur une petite machine sans GPU ;
— **Coût financier limité** : éviter d'avoir à maintenir des modèles lourds ou des serveurs dimensionnés pour de nombreux appels LLM simultanés.

### 6.2 Comparaison avec une approche classique

Une approche « full IA » classique pourrait consister à :

— déployer un modèle de type BERT ou GPT en local ;
— stocker l'intégralité des textes scrappés dans un moteur de recherche vectoriel (type FAISS ou similar) ;

— appeler systématiquement le LLM pour chaque question.

Cette stratégie présente plusieurs inconvénients dans notre contexte :

— besoin d'une **machine puissante** (GPU, RAM importante) ;
— augmentation des **coûts d'hébergement** et de maintenance ;
— plus grande **fragilité** en cas de panne de l'infrastructure IA.

À l'inverse, notre approche low-cost :

— utilise une base de FAQs **compacte et maîtrisée** ;
— s'appuie sur des embeddings **extrêmement légers** et calculables côté navigateur ;
— ne fait appel au LLM distant qu'en mode Online, et seulement pour **enrichir** des réponses déjà bien cadrées par le RAG ;
— garantit **deux modes totalement autonomes** (Offline et Hybride) même en absence de LLM ou de connexion.

### 6.3 Bénéfices pédagogiques

Cette architecture est également intéressante dans un cadre pédagogique :

— elle illustre les **principes de RAG** de manière simple ;
— elle montre comment concevoir une IA **progressive**, s'adaptant à la qualité du réseau ;
— elle met en avant des techniques **frugales** (hash-based embeddings) pertinentes pour des contextes à faibles ressources.

## 7 Sécurité et protection des données

Quelques principes simples sont mis en oeuvre :

— l'assistant ne stocke pas de données personnelles sensibles ;
— les conversations peuvent être effacées côté client ;
— la clé `OPENROUTER_API_KEY` est conservée dans les variables d'environnement du backend et n'est jamais exposée au frontend ;
— le déploiement en production doit se faire en **HTTPS** pour chiffrer les échanges entre le frontend, le backend et l'API OpenRouter.

## 8 Déploiement et perspectives

Le frontend est déployable sur n'importe quel hébergement de fichiers statiques, et le backend peut tourner sur un petit serveur FastAPI/Uvicorn.

Pistes d'évolution :

— enrichir la base de FAQ à partir des questions réellement posées pendant l'événement ;
— ajouter un mode « debug » dans l'interface pour afficher les sources et les scores de similarité ;
— expérimenter d'autres schémas d'embeddings frugaux (TF–IDF compact, hashing pondéré, etc.) ;
— ajouter une interface d'administration pour éditer les FAQs sans toucher au code.

## Conclusion

Cet assistant IA low-cost démontre qu'il est possible de combiner :

— une **bonne expérience utilisateur** (chat simple, PWA, bilingue) ;
— une **IA utile** (FAQ structurée, RAG, LLM) ;

— et des **contraintes réalistes de coût et d'infrastructure**.

L'approche multi-modes (Offline / Hybride / Online) permet de s'adapter à la réalité du terrain : les utilisateurs bénéficient toujours d'un niveau de réponse minimum, et, lorsque la connexion le permet, d'une réponse plus riche générée par un LLM distant.