Evaluation Certifiante

Rapport d’architecture technique : solution automatisée d’analyse de tweets SAV à l’aide de LLM

Bloc 2 - C2.3

Réalisé par :

SGHIOURI Mohammed
SOBGUI Ivan Joel
BOTI Armel Cyrille
BEN LOL Oumar
DIVENGI KIBANGUDI BUNKEMBO Nagui
ELOUMOU MBOUDOU Pascal Aurele

Date : 19 Novembre 2025

Tables de matières

1. Architecture Technique Globale

1.1. Contexte et Objectifs

La solution **Dallosh Analysis** a pour objectif principal de transformer un flux massif de retours clients (tweets exportés en CSV, environ 5 000, potentiellement plus à terme) en informations décisionnelles. Le but n'est pas la réponse client, mais l'**intelligence métier** : catégoriser, analyser la tonalité et extraire les motifs précis d'insatisfaction (pannes, problèmes de réseau, facturation, etc.).

L'architecture est conçue pour être **asynchrone et événementielle** (*Event-Driven*). Lorsqu'un un utilisateur dépose un fichier CSV via l'interface, le traitement lourd est immédiatement délégué à des processus d'arrière-plan (workers), ce qui garantit une haute réactivité de l'interface utilisateur. Cette approche permet de maintenir une expérience utilisateur fluide tout en traitant des volumes importants de données en arrière-plan.

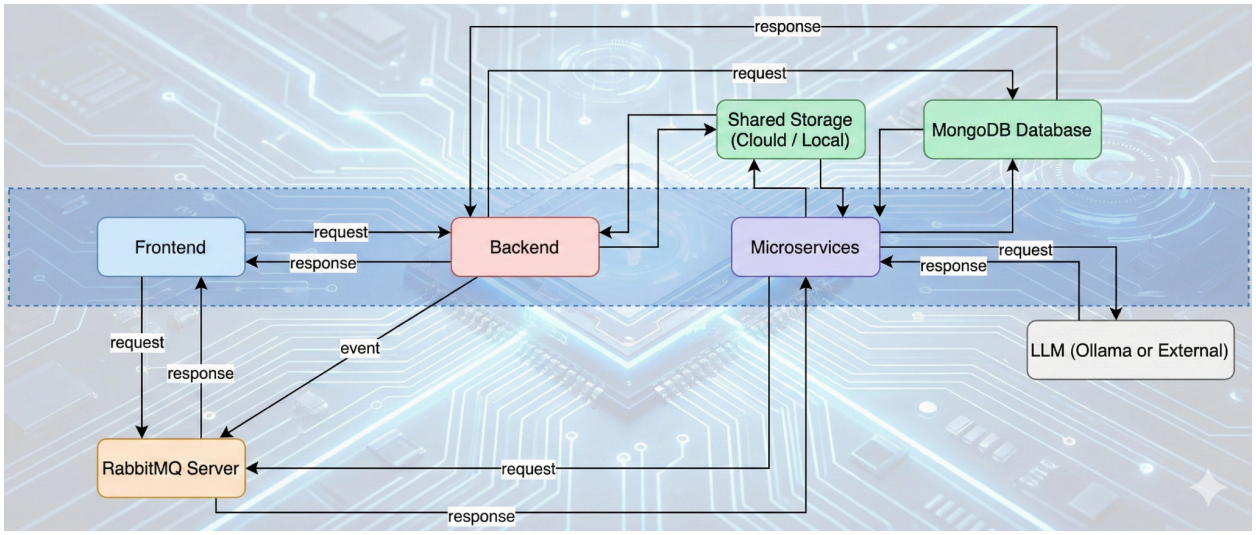
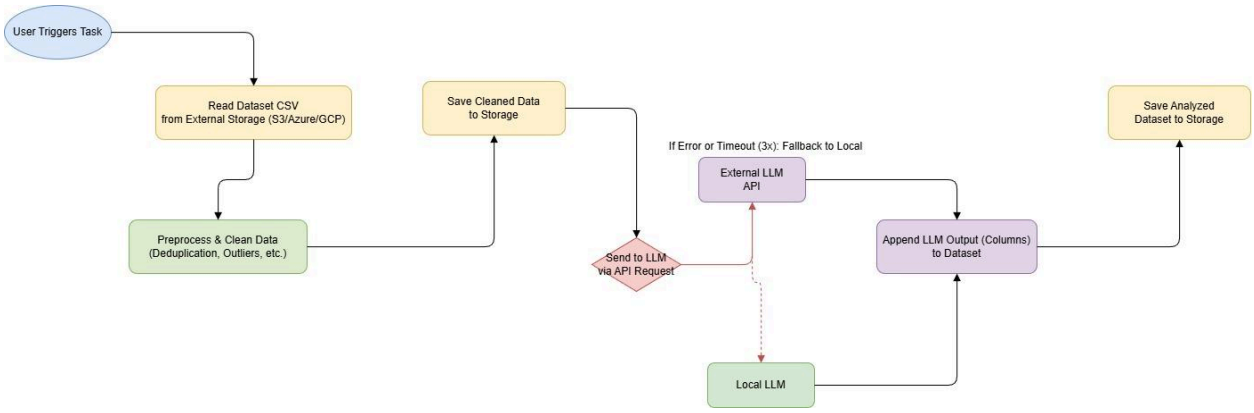


Figure 1 : Architecture technique

1.2. Cartographie des Composants et Pipeline de Traitement

Le système est structuré autour de quatre blocs fonctionnels, orchestrés par une file de messages (message broker) pour garantir la résilience et la scalabilité.

Schéma illustré du pipeline de traitement :



Le pipeline de traitement suit la séquence logique : **Chargement CSV → Nettoyage → Traitement par LLM → Classification → Visualisation**.

Identification des modules et flux de données :

Module (Zone Fonctionnelle)	Rôle dans le Pipeline	Composants Clés
Interface (Frontend/Backend)	**Chargement CSV** et **Visualisation** finale.	Next.js, Express.js (API Gateway)
Orchestration (Broker)	Gestion des files d'attente (file d'ingestion et file de résultats).	RabbitMQ
Analyse (Prétraitement & LLM)	**Nettoyage**, **Traitement par LLM**, **Classification** (extraction des motifs).	Python, Celery, Pandas, Tiktoken, Routeur IA (Gemini/Llama)
Stockage & État	Stockage des résultats structurés et agrégés pour la visualisation.	MongoDB

Le pipeline séquentiel est le suivant :

1. **Chargement CSV :** Dans un premier temps, l''utilisateur dépose le fichier via le Frontend. L'API Gateway (Express.js) reçoit et valide la requête.

2. **Délégation & Ingestion :** Dans un second temps, l'API place les tâches d'analyse, correspondant aux références des tweets, dans la file de messages (RabbitMQ).
3. **Nettoyage & Prétraitement :** Les Workers Python (Celery) récupèrent les tâches depuis la file de messages. `Pandas` est utilisé pour la manipulation du CSV. `Tiktoken` mesure le coût en tokens avant l'appel LLM.
4. **Traitement par LLM & Classification :** Le **Routeur Hybride IA** est sollicité pour la classification et l'extraction des informations (motif précis, tonalité). Le LLM renvoie les données structurées.
5. **Stockage :** Les résultats structurés sont stockés dans MongoDB.
6. **Visualisation :** Le Frontend lit les données de MongoDB pour afficher le tableau de bord en temps réel (KPIs, répartition des motifs).

2. Technologies Utilisées et Rôles

Domaine	Technologie Retenue	Rôle et Justification
LLM et Méthode d'Appel	**Google Gemini 2.5 Pro (API) Ollama + Llama 3.2 (Modèle Local)**	Le modèle Gemini est l'outil principal (API externe) pour sa performance et sa fenêtre de contexte. Llama 3.2 (via Ollama) est le modèle de secours pour la résilience.
Outils de Traitement	**Pandas & Tiktoken (Python)**	**Pandas** est l'outil standard pour le nettoyage des données CSV (prétraitement). **Tiktoken** est utilisé pour le comptage des tokens, optimisant le *batching* vers l'API.
Outils de Visualisation	**Next.js 16 (React 19) & Recharts**	Solution d'interface utilisateur custom, fournissant une visualisation interactive et en temps réel (difficile avec des outils BI classiques comme Power BI/Tableau).
Environnement Technique	**Python 3.11 & Celery**	**Python** est l'environnement Data Science pour le traitement. **Celery** assure la distribution des tâches lourdes sur les workers (cloud/local), permettant la conteneurisation.
Orchestration	**RabbitMQ**	Message broker pour la gestion asynchrone des tâches.
Stockage	**MongoDB**	Base de données NoSQL pour stocker les résultats d'analyse (motifs, tonalité).

3. Comparaison et Justification des Choix

3.1. Stratégie LLM

Notre stratégie repose sur une approche hybride combinant Gemini, en priorité, et Llama en local

Pourquoi ce modèle LLM plutôt qu'un autre ?

- **Gemini 2.5 Pro :** Ce modèle a été retenu comme choix par défaut en raison de sa haute performance, de sa grande fenêtre de contexte qui permet la scalabilité pour le batching, et de sa rapidité d'exécution, garantissant un traitement rapide du volume mais également pour le coût attrayant.
- **Llama 3.2 (Fallback) :** Comme les LLM sont performants, nous avons besoin d'assurer la continuité du service sans coût supplémentaire en cas de panne ou de dépassement de quota de l'API externe. C'est ainsi que nous avons porté naturellement notre choix sur ce modèle qui apporte la **résilience** et la **souveraineté**.

3.2. Solution de Visualisation :

Custom (Next.js/Recharts) vs BI (Power BI/Tableau)

Pourquoi telle technologie de dashboard ?

Le choix de **Next.js/Recharts** plutôt qu'un outil BI standard est justifié par le **besoin d'interactivité en temps réel**. L'architecture asynchrone permet à l'interface de recevoir l'état d'avancement du traitement via RabbitMQ et d'afficher les résultats à la volée.

Cette intégration de données dynamiques apparaît complexe, voire impossible, avec les outils BI conçus pour des bases statiques. Par conséquent, une solution custom offre une flexibilité supérieure pour répondre aux exigences spécifiques du projet.

3.3. Scalabilité, Coût, Accessibilité, Support Communautaire

Critère	Justification du Choix
Scalabilité	L'architecture Microservices/Event-Driven (RabbitMQ + Celery) permet un **Scale-out horizontal** simple : l'ajout de Workers Python est la seule action nécessaire pour gérer l'augmentation de volume (5 000 →50 000 tweets).
Coût	Le choix d'une interface custom (Next.js) évite les **coûts de licence par utilisateur** des outils BI.
Accessibilité/Support	L'utilisation de **Python** (Pandas, Celery) et de **JavaScript** (Node.js, Next.js) s'appuie sur les écosystèmes les plus robustes et les plus documentés, assurant un **support communautaire** maximal et une facilité d'intégration.

4. Contraintes et Alternatives

4.1. Gestion du volume de données

(5 000 tweets maintenant, potentiellement 50 000 plus tard)

L'architecture est nativement conçue pour la **scalabilité horizontale** (*Scale-out*). Pour passer de 5 000 à 50 000 tweets sans compromettre la performance, la solution prévoit l'ajout simple de conteneurs **Worker Python** supplémentaires. Le message broker (RabbitMQ) assure alors automatiquement la distribution de la charge entre les différents nœuds.

De plus, l'efficacité est garantie par la **Stratégie de "Token Bucket"** : l'agrégation des tweets en lots (batches) réduit le nombre d'appels API, assurant une performance constante face à l'augmentation du volume.

4.2. Limites API

La dépendance à l'API Gemini introduit des contraintes de quota, de coût et de vitesse qui sont gérées par les dispositifs suivants :

- **Quota & Vitesse :** Le **Routeur Hybride IA** bascule le traitement sur le modèle local (Llama 3.2) en cas de dépassement de quota ou de latence élevée de l'API externe. Le framework **Celery** permet également de réguler le taux d'appels (*Rate Limiting*) en mettant les requêtes en attente, assurant l'intégrité des données.
- **Coût :** La **Stratégie de "Token Bucket"** minimise le coût en maximisant l'utilisation de la fenêtre de contexte de Gemini, réduisant le nombre de transactions HTTP facturées.
- **Alternative :** Pour une indépendance totale et une meilleure maîtrise des coûts à très grande échelle (500 000+), une alternative serait l'investissement dans une carte GPU dédiée pour le *Worker* local, associée à un LLM Open Source léger et performant (ex: Phi-3-mini).

5. Conclusion

L'architecture Dallosh Analysis présente des caractéristiques de performance et de stratégie qui méritent d'être soulignées combinant l'efficacité d'une API externe (Gemini) avec la résilience d'un modèle local (Llama), offrant ainsi le meilleur compromis entre performance, coût et résilience.

Le découplage des services via RabbitMQ et l'optimisation du batching par tokens assurent que la solution peut facilement absorber une charge de travail dix fois supérieure à la demande initiale sans compromettre l'expérience utilisateur. Cette capacité d'évolution constitue un atout majeur pour l'adaptation future aux besoins croissants de l'organisation.

Enfin, il apparaît que les choix technologiques retenus permettent de répondre aux contraintes identifiées tout en préservant la flexibilité nécessaire à l'évolution du système. D’où, la stratégie hybride adoptée offre une base solide pour le déploiement en production et l'adaptation aux évolutions futures des besoins métier.