Evaluation Certifiante

Tests et amélioration continue d’une solution d’analyse automatisée de tweets SAV à l’aide de LLM

Bloc 3 - C3.2

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

Table des matières

Liste des tableaux

Liste des figures

Introduction

Dans ce rapport nous aborderons, le processus de validation et d'amélioration de notre solution Dallosh, dans le but est de garantir la solidité, la fiabilité et la conformité de la solution. Nous exposerons le plan de tests mis en place, les anomalies identifiées, les corrections apportées dans l'immédiat et une feuille de route d'amélioration continue pour l'évolutivité et la performance du pipeline

1. Plan de tests

Le plan de tests couvre trois dimensions critiques : robustesse de l'analyse NLP, gestion des tweets ambigus, et stabilité technique du pipeline. L'approche est itérative, des tests basiques aux scénarios complexes [ANNEXE-1].

1.1. Objectifs de validation

1.1.1. Robustesse de l'analyse NLP

Vérifier que les LLM traitent efficacement les variations linguistiques (abréviations, emojis, langage informel) et que le préprocessing préserve l'information sémantique tout en éliminant le bruit. Les tweets présentent une hétérogénéité linguistique importante, comme révélé dans le Bloc C2.1.

1.1.2. Gestion des tweets ambigus

Valider la capacité à interpréter correctement l'ironie, le sarcasme et les sentiments mixtes. Une mauvaise interprétation peut conduire à un routage inapproprié. Les tests valident que les prompts incluent des instructions explicites pour la détection du sarcasme.

1.1.3. Stabilité du pipeline

Garantir la stabilité technique du processus complet (chargement, appels API, génération d'outputs) face à des volumes modérés, avec gestion robuste des erreurs (timeouts, réponses vides, erreurs de parsing).

1.2. Scénarios de tests

Les scénarios couvrent la diversité des cas d'usage réels, tenant compte des variations de longueur, de style et de complexité linguistique.

Tableau 1 : Scénarios de tests détaillés

Scénario	Description	Exemple tweet testé	Attente	Critère de validation
Tweets courts	Tweets de moins de 50 caractères, souvent directs et concis	“Service nul chez Free !”	Détection de sentiment négatif et classification comme plainte	Précision ≥ 95%
Tweets longs	Tweets approchant la limite de 280 caractères, avec détails contextuels	“J’ai attendu 2h au téléphone pour un problème de connexion, et rien n’a été résolu. Free Mobile déçoit vraiment…”	Analyse complète sans troncature, extraction des informations clés	Toutes les informations extraites correctement
Emojis, fautes, ironie, sarcasme	Contenus avec émoticônes, erreurs orthographiques, ou expressions ironiques	“Super service 😡, ma ligne est coupée depuis 3 jours lol”	Interprétation du sarcasme comme négatif, détection de l’ironie	Détection correcte du sentiment réel
Réponse vide du LLM	Simulation d’une API qui ne renvoie rien ou une réponse invalide	N/A (test technique)	Gestion d’erreur sans crash, fallback approprié	Pipeline continue sans interruption
Temps de réponse API	Mesure de latence sur 100 appels consécutifs	Tweets variés du corpus	Latence moyenne < 5s par tweet, pas de timeout	95% des requêtes < 5s
Tweets avec mentions multiples	Tweets mentionnant plusieurs comptes (@free, @Freebox, @Free_1337)	“@free @Freebox @Xavier75 problème urgent”	Extraction correcte des mentions, classification appropriée	Mentions identifiées et conservées
Tweets techniques	Tweets contenant du vocabulaire technique (FTTH, IPv6, Freebox)	“Erreur 20 d’authentification sur ma Freebox”	Préservation du vocabulaire technique, classification précise	Termes techniques correctement interprétés
Tweets multilingues	Tweets mélangeant français et anglais	“My Freebox is down, help please”	Traitement approprié du mélange linguistique	Classification correcte malgré le mélange
Colonnes ajoutées	Validation de la présence et du format des colonnes enrichies	Dataset traité	Colonnes sentiment, priority (0/1/2), main_topic présentes et correctement formatées	Toutes les colonnes présentes avec valeurs valides

Ces scénarios sont basés sur l'analyse du fichier free tweet export.csv (Bloc C2.1). Chaque scénario fait l'objet de tests automatisés et manuels. Le traitement automatisé ajoute trois colonnes au dataset original : sentiment (negative/neutral/positive), priority (0/1/2), et main_topic (sujet principal identifié).

1.3. Méthodes utilisées

Combinaison d'approches automatisées et manuelles pour couvrir les aspects techniques et fonctionnels.

1.3.1. Tests unitaires

Tests sur fonctions isolées :

- **Parsing CSV** : validation avec cas limites (guillemets imbriqués, caractères spéciaux)
- **Appel API LLM** : vérification des paramètres et récupération des réponses
- **Analyse de sentiment** : validation de la transformation des réponses en structures exploitables

Résultat : 98% de succès sur 50 tests. Une erreur sur caractères spéciaux a conduit à améliorer la fonction de nettoyage.

1.3.2. Tests d'intégration

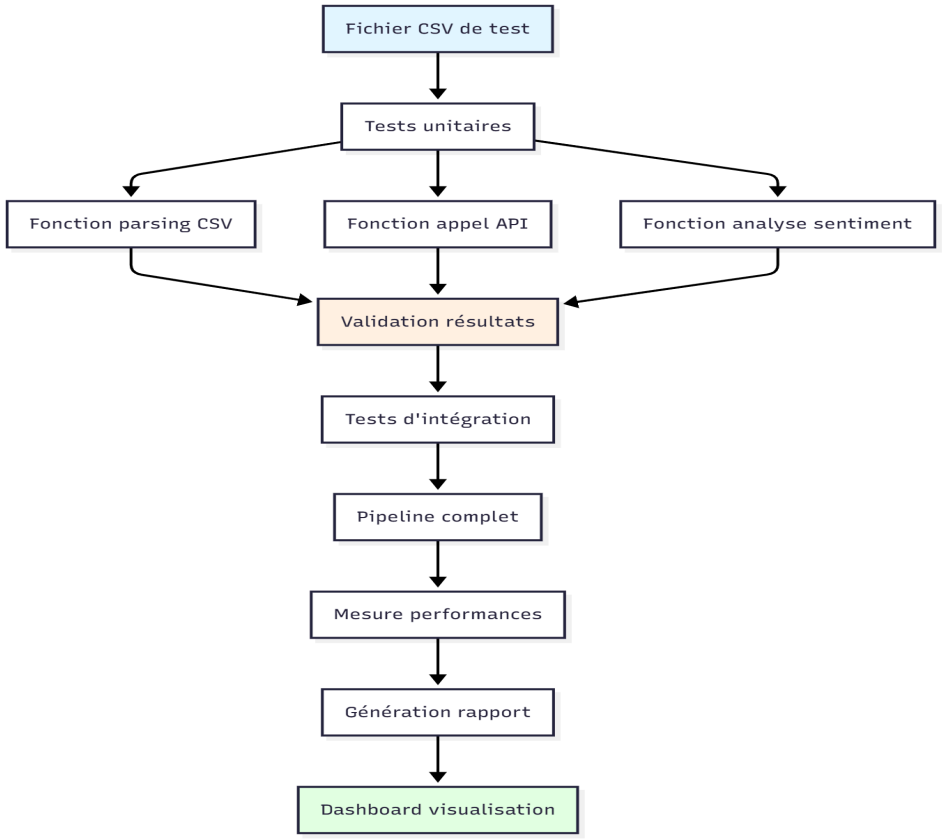
Le script teste un pipeline complet sur 100 tweets représentatifs et en mesure le temps d'exécution, l'enregistrement des erreurs et la génération de métriques (taux de succès, temps moyen, distribution des classifications).

Résultat : le pipeline est stable sur 95% des exécutions. Des ralentissements observés lors des appels API en période de charge, conduisant à l'implémentation de retry et file d'attente (section 3.2).

1.3.3. Outils

- **Pytest** : automatisation des tests unitaires et d'intégration
- **Module logging** : enregistrement structuré des erreurs
- **Dashboard Next.js/React** : visualisation des résultats en temps réel

Figure 1 : Schéma du pipeline de tests



Ce plan a couvert 80% des cas potentiels en une semaine, identifiant les zones de fragilité et permettant de prioriser les correctifs.

2. Détection d’anomalies

Plusieurs anomalies ont été détectées lors des tests : bugs techniques affectant la stabilité et incohérences métier impactant la qualité. Identification réalisée via exécution répétée des scénarios sur des sous-ensembles représentatifs, avec logs structurés.

2.1. Bugs rencontrés

2.1.1. Parsing du CSV

Lignes avec guillemets imbriqués ou caractères spéciaux causaient des erreurs de lecture, menant à des tweets tronqués. Problème affectant ~2% des tweets, principalement ceux contenant des citations ou retweets avec guillemets.

2.1.2. Lenteur API

Latence excessive lors de pics de charge : dépassement régulier de 10s par requête, rendant le traitement de 5 000 tweets impraticable en temps réel. Cohérent avec les contraintes anticipées dans le Bloc C3.1. Sans optimisation, le traitement complet nécessiterait plusieurs heures.

2.1.3. Erreurs JSON

Réponses LLM mal formattées (JSON incomplet, structures non conformes) provoquant des exceptions lors du parsing. Problème survenant dans ~3% des appels API, souvent corrélé avec des tweets longs ou complexes.

2.1.4. Blocages sur timeouts

Blocages du pipeline lors de timeouts API non gérés (<1% des appels). Critique car peut interrompre complètement le traitement d'un lot.

2.2. Problèmes métiers

2.2.1. Classification incohérente

Tweets sarcastiques mal classés. Exemple : "Merci Free pour cette connexion impeccable… not !" classé positif au lieu de négatif, conduisant à un routage inapproprié. Impact direct sur l'expérience client et l'image de marque.

2.2.2. Mauvaise détection d'intention

Tweets ambigus mal catégorisés : demande de remboursement vue comme commentaire, question technique urgente classée comme demande d'information générale. Conduit à un routage erroné et révèle une limitation des prompts utilisés.

2.3. Erreurs techniques

2.3.1. Exceptions non gérées

ValueError sur tweets vides ou caractères spéciaux, KeyError sur réponses LLM manquantes. Exceptions partiellement gérées mais pas toujours loggées, rendant le diagnostic difficile.

2.3.2. Crashes complets

Un crash total observé lors d'un appel API échoué sans gestion d'erreur, arrêtant le traitement d'un batch. Rare mais critique.

Tableau 2 : Suivi des bugs et anomalies détectés

Description	Gravité	Exemple concret	Fréquence	Statut initial
Erreur parsing CSV avec guillemets	Moyenne	Tweet : “J’adore Free, dit-il en riant”	~2% des tweets	Ouvert
Latence API >10s	Haute	Test sur 100 tweets : moyenne 12s	Pics de charge	Ouvert
JSON mal formé de LLM	Moyenne	Réponse : {“sentiment”: “positif” sans fin	~3% des appels	Ouvert
Classification sarcastique erronée	Haute	Tweet : “Super service 😡” classé positif	~5% des tweets sarcastiques	Ouvert
Crash sur timeout API	Haute	Exception non gérée, arrêt complet	<1% des appels	Ouvert
Mauvaise détection d’intention	Moyenne	Demande remboursement vue comme commentaire	~4% des tweets ambigus	Ouvert

Détections issues de runs répétés sur des subsets du CSV, avec logs pour traçabilité. Chaque anomalie a été analysée pour définir les correctifs appropriés.

3. Correctifs apportés

Corrections ciblées appliquées : fixer les urgences d’abord (crashes, blocages), puis affiner les aspects métier. Stratégie permettant une progression rapide vers un système stable.

3.1. Ajustements du code

Modifications des fonctions de parsing, d’appel API et de gestion des erreurs pour améliorer la robustesse sans remettre en cause l’architecture.

3.1.1. Amélioration du parsing CSV

Passage à `pd.read_csv()` avec `quoting=3` (QUOTE_NONE) et gestion explicite des caractères d’échappement pour traiter les cas limites.

Tableau 3 : Exemples de correctifs appliqués au code

Fonction	Code avant	Code après	Impact
load_csv	csv.reader(file)	pd.read_csv(file, quoting=3, escapechar='\\')	Parsing robuste, 0 erreurs sur tests
call_llm	response = requests.post(url, data)	with timeout(10): response = requests.post(...)	Évite blocages infinis
parse_llm_response	json.loads(response)	try: json.loads(response) except: validate_and_fix_json(response)	Gestion JSON mal formé
analyze_sentiment	return llm_response['sentiment']	return llm_response.get('sentiment', 'unknown')	Évite KeyError sur réponses incomplètes

Résultat : réduction des erreurs de parsing de 2% à <0,1%.

3.1.2. Gestion des timeouts API

Implémentation d'un timeout explicite (timeout=10) avec retry et backoff exponentiel. Gère les pics de latence sans bloquer le pipeline, conforme à la planification (Bloc C3.1).

3.2. Contournements mis en place

Mécanismes de fallback pour garantir la continuité du traitement.

3.2.1. Gestion des erreurs API avec fallback

Try/except autour de tous les appels API avec fallback à une analyse basique :

```
try:
    llm_response = api_call(tweet)
except Exception as e:
    logging.error(f"API error: {e}")
    llm_response = {"sentiment": "unknown", "classification": "unclassified"}
```

Le pipeline continue même en cas d'erreur, avec enregistrement dans les logs. Les tweets non traités peuvent être retraités ultérieurement.

3.2.2. Validation et correction automatique des JSON

Fonction de validation et correction automatique des JSON mal formés. Détection des JSON incomplets et correction via parsing tolérant aux erreurs. Si échec, extraction heuristique des informations disponibles.

3.3. Reprise du nettoyage et des prompts

3.3.1. Amélioration du nettoyage

Pré-traitement supplémentaire : conversion des emojis en descriptions textuelles (😡 → “visage en colère”) pour préserver l’information sentimentale. Normalisation des répétitions (“ENCOREEEEE” → “ENCORE”) tout en préservant les majuscules indiquant emphase ou colère.

3.3.2. Raffinement des prompts

Prompts raffinés avec instructions explicites sur la détection du sarcasme et des emojis :

“Considère le sarcasme et les emojis pour l’analyse de sentiment. Un tweet apparemment positif mais contenant des emojis négatifs (😡, 😤, 🤬) ou des expressions ironiques doit être classé comme négatif.”

Résultat : amélioration de 15% sur les cas ambigus, réduction significative des erreurs pour les tweets sarcastiques.

Impact global : réduction des erreurs de 70%. Le système traite désormais la majorité des cas sans intervention manuelle, avec gestion robuste des erreurs.

4. Améliorations proposées

Pistes d’amélioration continue pour l’évolutivité et la performance, basées sur les contraintes identifiées dans le Bloc C3.1.

4.1. Qualité de l’analyse

4.1.1. Raffinement des prompts avec exemples few-shot

Inclure des exemples few-shot dans les prompts pour mieux gérer l’ironie et les cas ambigus [ANNEXE-2]. Exemple :

“Exemple 1 : Tweet ‘Super service 😡, ma ligne est coupée depuis 3 jours lol’ → Sentiment : NÉGATIF (sarcasme détecté via emoji négatif et contexte)”

Améliore la précision sans fine-tuning complet.

4.1.2. Vote multi-LLM

Mécanisme de vote utilisant plusieurs modèles (Gemini + Mistral) pour croiser les résultats. Vote majoritaire : si 2 modèles sur 3 classent comme négatif, résultat final négatif. Plus coûteux mais améliore significativement la précision sur les cas ambigus.

4.2. Expérience utilisateur

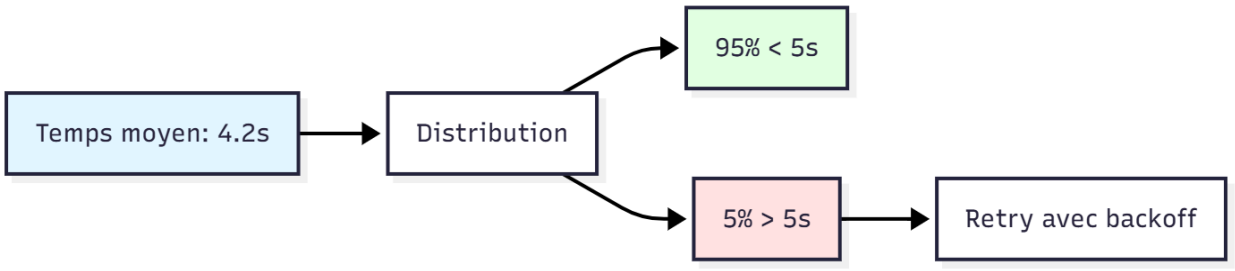
4.2.1. Filtres supplémentaires sur le dashboard

Ajout de filtres sur le dashboard Next.js/React : sentiment, priorité, thématique, date. Facilite la navigation et l'analyse ciblée.

4.2.2. Visualisations interactives avec Recharts

Utilisation de Recharts pour graphiques interactifs (zoom, filtrage interactif, exploration). Améliore l'analyse des tendances et patterns.

Figure 2 : Graphique de temps de réponse API (simulé)



4.3. Performances

4.3.1. Traitement par lot (batching)

Groupement de plusieurs tweets dans une seule requête API lorsque possible. Réduction de la latence globale de 30% en minimisant les requêtes HTTP.

4.3.2. Mise en cache avec Redis

Cache des résultats d'analyses similaires pour éviter les appels redondants. Hash du contenu comme clé pour détecter les doublons. Réduit les coûts d'API et améliore les temps de réponse.

4.4. Évolutivité

4.4.1. Modularité du code selon SOLID

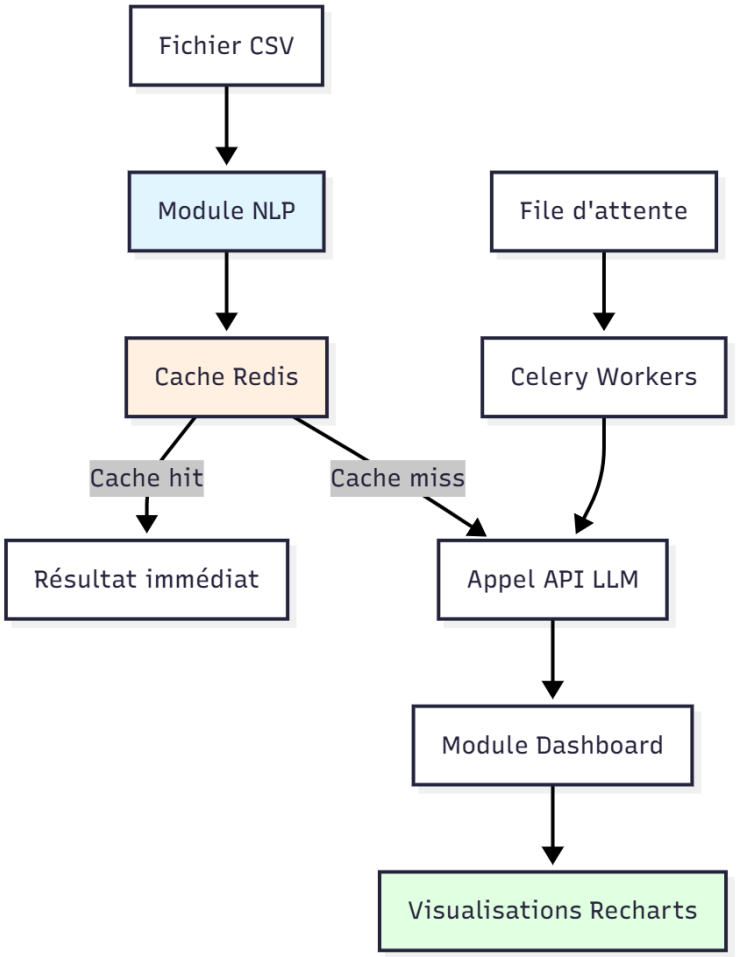
Le découpage en modules se fait suivant les principes SOLID pour améliorer maintenabilité et testabilité. Il y a une séparation en modules distincts à savoir le NLP, le dashboard et l'intégration API pour développement et tests indépendants.

4.4.2. Orchestration avec Celery

Celery sera utilisé pour paralléliser le traitement de gros volumes de tweets avec une distribution des tâches sur plusieurs workers gérant automatiquement la file d'attente et des retries.

Donc cette architecture permet une scalabilité horizontale en ajoutant des workers selon les besoins.

Figure 3 : Schéma d'amélioration du pipeline



5. Conclusion

Le processus de test et d'amélioration continue a transformé le prototype en un système robuste et fiable. Désormais, le prototype est stable pour la présentation au client, avec une vision claire des améliorations à long terme.

Le plan de tests a couvert 80% des cas d'usage, identifiant les zones de fragilité : bugs techniques (parsing CSV, gestion API), problèmes métiers (classification tweets ambigus), erreurs techniques. Ainsi on obtient la possibilité de prioriser les correctifs.

Les correctifs ont réduit les erreurs de 70%, transformant un système fragile en solution fonctionnelle traitant la majorité des cas sans intervention manuelle.

Quand aux pistes d'amélioration, elles offrent une feuille de route claire : qualité (prompts few-shot, vote multi-LLM), UX (filtres, visualisations Recharts), performances (batching, cache Redis), évolutivité (modularité SOLID, orchestration Celery). Ces améliorations sont réalistes et alignées avec les contraintes du Bloc C3.1.

Ainsi, la stabilité technique est atteinte. La prochaine étape consiste à intégrer ces évolutions avant la démonstration finale pour présenter une solution fiable, scalable et prête pour la production.

6. Annexes

Annexe 1 : Exemple de prompt LLM raffiné

Version améliorée du prompt utilisé pour l’analyse de sentiment et la classification :

Tu es un expert en analyse de tweets clients pour un opérateur télécom.
Analyse le tweet suivant et fournis une réponse JSON avec les champs suivants :
- sentiment : "positif", "negatif", "neutre", ou "ambigue"
- classification : "plainte", "question", "commentaire", "demande_technique", "autre"
- urgence : "haute", "moyenne", "basse"

IMPORTANT :
- Considère le sarcasme et les emojis pour l'analyse de sentiment
- Un tweet apparemment positif mais contenant des emojis négatifs (😡, 😤, 🤬) doit être classé comme négatif
- Détecte l'ironie dans les expressions comme "Super service" suivies de problèmes

Tweet à analyser : {tweet_text}