D-ScrAibe

Combiner les avantages du papier et du numérique

Introduction

Contexte

Chez RTE, les opérateurs en salle de dispatching sont chargés de la surveillance et du pilotage de l'unique réseau national français de transport d'électricité. Répartis sur les différents niveaux du réseau et sur les différentes zones du territoire métropolitain français, les dispatchers font face à de grandes quantités d'appels et à un nombre parfois intense de sollicitations. La salle de dispatching est ainsi un espace de travail en équipe confronté à des situations d'urgence.

Pour augmenter la résilience de l'équipe de dispatching face à ces situations d'urgence, RTE développe Storylines, un outil qui vise à construire une manière partagée par tous les dispatcheurs de voir en temps réel l'état du réseau. Afin de construire cette vision partagée du réseau, il faut récolter, en temps réel, les données relatives à l'évolution du réseau. Cependant, les dispatcheurs, faisant déjà l'objet de nombreuses sollicitations, ne peuvent pas renseigner, lors de leur quart, ces informations sur l'outil informatique. Il s'agit donc de capter de manière automatique l'ensemble des informations importantes reçues par l'opérateur.

Un moyen simple est de se baser sur un support déjà largement utilisé par les dispatcheurs : des feuilles de prise de notes. Pour organiser le flux de données continu auquel il fait face, le dispatcher prend régulièrement en note les informations qu'il juge importantes et surtout qu'il pense intéressantes pour le futur (la suite de son quart ou le quart suivant).

Notre projet s'inscrit donc dans ce cadre : dans l'idée de fournir de la matière à l'outil Storylines, nous avons développé un outil de numérisation automatique et en temps réel des notes manuscrites des opérateurs.

Enjeux

Cet outil de numérisation des notes manuscrites des opérateurs doit respecter un certain nombre de contraintes pour optimiser son adoption possible en salle de dispatching.
Face à un environnement dense en informations et demandant une importante attention, le dispatcheur ne doit pas ressentir la présence de l'outil et avoir à s'assurer de son bon fonctionnement.

Il faut que la tâche du dispatcher soit identique avec ou sans l'outil, si ce n'est plus facile. L'outil doit ainsi être **transparent** du point de vue du dispatcher.

Un point important dans les données récoltées et transmises à Storylines est l'horodatage, afin de pouvoir retracer l'évolution du réseau. La numérisation des notes doit ainsi se faire en **temps réel**, ou quasi réel (avec un décalage de l'ordre de la minute ou de la dizaine de minutes).

Enfin, un certain niveau de **robustesse** est attendu, afin que les données affichées dans l'outil Storylines soient fiables (réelles et non manquantes).

Aperçu de la solution

Afin de maximiser la quantité d'informations captées, D-ScrAibe se base sur deux canaux différents :

- Un canal vidéo pour capter les notes manuscrites et leur évolution.
- Et un canal audio pour capter les phrases prononcées par le dispatcher.

En sortie de chacun de ces canaux, nous obtenons un texte nettoyé d'information brute que l'on considère intéressante à conserver. Ces informations sont ensuite traitées de manière similaire. Ces textes sont annotés, c'est-à-dire qu'on en extrait les mots-clés importants pour notre cas d'usage et on les catégorise (lieu, acteur, procédure, …). Ils sont ensuite envoyés ainsi augmentés à l'application de retour utilisateur, qui permet d'afficher de manière chronologique les informations captées.

Canal vidéo

La captation vidéo se fait à l'aide d'une caméra placée sur un trépied à 80 cm au-dessus de la table où l'opérateur prend ses notes. Lors du projet, pour des raisons de simplicité et de qualité photo, et travaillant dans l'écosystème Apple, nous avons utilisé pour caméra un iPhone connecté en Bluetooth au Macbook sur lequel le programme était lancé. L'intérêt ici est la prise en compte native de l'IPhone comme périphérique vidéo par le Macbook (lorsqu'il est connecté, celui-ci est considéré comme la source 0, à la place de la webcam lorsqu'il n'est pas connecté).

Si cette solution n'est pas disponible, des scripts permettant l'utilisation d'une Raspberry Pi agrémentée d'une caméra sont disponibles dans le code source du projet (nous détaillerons ceux-ci par la suite).

La structure de captation vidéo des notes manuscrites se composent de plusieurs composants que nous allons développer ci-dessous :

- Initialisation du hardware de captation vidéo (si nécessaire).
- Détection de feuilles dans les images captées.
- Rognage et correction de la perspective et rotation afin d'obtenir une image rectangulaire et verticale.
- Comparaison de l'image obtenue de la feuille avec l'historique des feuilles déjà enregistrées pour éviter les doublons.
- Reconnaissance du texte manuscrit (HTR).
- Isolement des informations nouvelles.

Hardware vidéo

Dans le cas où l'utilisation d'un Iphone couplé à un Macbook n'est pas possible, nous avons implémenté des scripts permettant la captation vidéo par une Raspberry Pi munie d'une caméra. A noter que des applications permettent de connecter un téléphone et un ordinateur, quel que soit leur système d'exploitation (voir DroidCam).

Dans la solution implémentée, pour des raisons de simplicité, la connexion avec la Raspberry Pi se fait en SSH. A noter que pour un passage à l'échelle, il serait plus avisé d'utiliser une API afin d'éviter les copies multiples de fichiers. Le lancement de la Raspberry Pi se fait entièrement depuis le script `src/raspberry/launch_rasp.py`. Il faut, au préalable, avoir ajouté la clé SSH de l'ordinateur maître utilisé aux clés SSH autorisées sur la Raspberry, et renseigner l'adresse IP de la Raspberry Pi et le nom d'utilisateur choisi sur le script `launch_rasp.py`.

Une fois le script lancé, l'ordinateur maître va se connecter en SSH à la Raspberry Pi et y transférer les scripts contenus dans `src/raspberry/paper_detection_raspberrypi/`, afin de s'assurer que c'est bien leur dernière version qui y est lancée. L'ensemble de ces scripts est une simple adaptation à la Raspberry des scripts décrits dans le paragraphe Détection des feuilles.

Ensuite est exécuté, sur la Raspberry Pi, le script `video_capture_raspberrypi.py`, qui est le script principal de la captation `vidéo/détection d'objets`. La caméra est alors en route, et une boucle est lancée pour surveiller, depuis l'ordinateur maître et en SFTP, l'arrivée de nouvelles images dans un dossier cible de la Raspberry. Lorsqu'une image est reçue dans ce dossier, elle est copiée sur l'ordinateur maître, supprimée de la Raspberry, et enfin passée en argument de la fonction add_data2db() qui représente la porte d'entrée du reste du pipeline, à partir de la reconnaissance HTR.

On s'aperçoit ainsi que les étapes de détection de feuille et de rognage/correction de perspective sont effectuées sur la Raspberry Pi.

L'arrêt du programme de capture vidéo sur la Raspberry est commandé par la présence ou non d'un fichier d'arrêt à l'adresse */home/*$user*/Documents/stop.txt*. Au début du script `launch_rasp.py`, il faut donc bien s'assurer que ce fichier d'arrêt n'est pas présent, et on recrée ce fichier sur la Raspberry Pi lorsque l'arrêt du script launch_rasp.py est demandé.

Détection des feuilles

Les images captées par la caméra sont analysées par un modèle de détection de feuilles. Nous avons implémenté deux modèles permettant ceci :

- Le premier se base sur une analyse des formes, des contours et des couleurs présents sur l'image.
- tandis que le deuxième s'appuie sur le modèle de détection d'objets pré-entraîné YOLOv11.

Modèle basé sur la détection de contour

Le premier modèle se base sur une analyse classique de l'image et part de l'hypothèse qu'une feuille est une forme quadrilatère blanche occupant un espace assez grand dans l'image.

Après un prétraitement assez simple de l'image (passage en niveaux de gris et filtre bilatéral pour réduire le bruit), elle est passée au modèle de détection de contours Canny.

Des transformations morphologiques permettent ensuite de fermer les différents contours détectés. Il ne reste plus qu'à effectuer une approximation polygonale de ces derniers puis à sélectionner seulement les quadrilatères valides en fonction de leur aire, couleur et convexité.

Cependant, bien que ce modèle soit simple et qu'il ait des performances satisfaisantes pour le cas d'une feuille mise en évidence, il est limité dans le cadre d'un environnement dense. En effet, il dépend fortement de la luminosité de l'environnement, du contraste entre la feuille et le fond (un arrière-plan blanc limite son efficacité tout comme un fond non uniforme) et de la forme de la feuille.

Afin d'essayer de pallier le problème de luminosité et du contraste, un script de configuration `config_shape_detector.py` permet de jouer sur les paramètres du modèle de détection de forme et d'avoir un retour visuel sur leurs changements.

Ces paramètres sont ceux du filtrage bilatéral :

- BILATERAL_D : diamètre du voisinage autour de chaque pixel (plus grand = lissage plus fort).
- BILATERAL_SIGMA_COLOR : sensibilité aux différences de couleur (plus élevé = couleurs plus mélangées).
- BILATERAL_SIGMA_SPACE : influence spatiale du filtre (plus élevé = zone de lissage plus large).

de la détection de contours par Canny :

- CANNY_THRESHOLD1 : seuil inférieur de détection.
- CANNY_THRESHOLD2 : seuil supérieur (plus élevé = moins de contours détectés).

de la fermeture morphologique des contours :

- MORPH_KERNEL_SIZE : taille du noyau (plus grand = contours plus épais).
- MORPH_ITERATIONS : nombre d'itérations (à augmenter si les contours restent discontinus).

de l'approximation polygonale :

- POLY_EPSILON_FACTOR : fraction du périmètre utilisée pour la simplification (faible = forme précise, élevé = forme simplifiée).

du filtrage des quadrilatères :

- MIN_AREA_RATIO : la forme doit représenter au moins X% de la surface de l'image.
- MAX_SATURATION : saturation maximale pour considérer une zone comme blanche.
- MIN_VALUE : luminosité minimale pour considérer une zone comme claire.

Lorsque la détection est satisfaisante, il retourne les valeurs choisies ce qui permet par la suite de les mettre à jour manuellement dans le script `shape_detector.py`.

Modèle basé sur la détection par YOLO

Afin de chercher un peu plus de robustesse face à une diversité de situations qu'il est possible de rencontrer, nous nous sommes penchés sur un modèle de deep learning. L'un des modèles de détection d'objet de référence aujourd'hui est YOLO. Nous avons donc choisi de fine-tuner le modèle YOLOv11 sur un dataset constitué d'images de feuilles tirées de la plateforme Roboflow.

YOLOv11 existe en modèle de détection (la sortie du modèle est une bounding box de l'objet détecté) et en modèle de segmentation (la sortie est un masque binaire distinguant les pixels appartenant à l'objet de ceux appartenant à l'arrière-plan).
Le modèle de segmentation que l'on a fine-tuné (`src/proc/detection_model/best-segment.pt`) donne beaucoup de pixels faux-positifs, ce qui compromet la tâche de rognage et de correction de la perspective.

Nous avons donc opté pour le modèle de détection d'objet, couplé à une segmentation objet/arrière-plan indépendante de YOLO, détaillé dans le paragraphe suivant Rognage et correction de perspective. De manière concrète, le modèle YOLO se lance depuis le fichier `yolo_tracker_photos.py`.

Ce fichier va lancer la caméra de l'IPhone, et traiter les photos prises par l'IPhone à une vitesse de 1 fps. Cette vitesse lente permet, dans l'absolu, de profiter de la résolution photo du périphérique, qui est meilleure que la résolution vidéo (dans les faits, la caméra de l'IPhone ne peut servir que de périphérique vidéo depuis un Macbook, donc ceci n'améliorera pas la définition de ce présent modèle, mais avec d'autres périphériques, cela pourrait fonctionner), et ne met pas en péril la détection de feuilles, le modèle YOLO ayant une bonne performance de rappel.

Ensuite, YOLO va détecter les éventuelles feuilles présentes. Pour chaque bounding box renvoyée par YOLO, nous faisons tourner le modèle de segmentation objet/arrière-plan qui permet de rogner proprement autour de la feuille. Un premier tri est effectué en analysant regardant si la couleur moyenne des pixels de la feuille est assez proche du blanc.

Ensuite, nous stockons les feuilles restantes dans un buffer, accompagnées de leur valeur le flou estimé. La valeur de flou de l'image est estimé en calculant la variance du laplacien de l'image, ce qui est censé donner de grandes valeurs pour des pixels changeant brusquement de couleur, c'est-à-dire avec des contours nets. Afin de se prémunir du cas où l'image envoyée est floue mais très colorée, ce qui renverrait une valeur également élevée, nous divisons l'image en 40*40 sous-blocs, nous calculons cette valeur de flou localement, et nous prenons comme valeur de sortie la valeur moyenne du flou local des 100 sous-blocs où le flou est maximal (donc où la variance du laplacien est minimal).

Lorsque le buffer contient plus de 5 images, le script envoie à la suite du pipeline l'image captée la moins floue, et le buffer est réinitialisé. Pour répondre au cas où plusieurs feuilles sont détectées par YOLO, un buffer est créé pour chaque feuille, et on vide les buffers dès qu'il y en a un qui atteint une longueur de 5. Les buffers sont également réinitialisés sans faire passer aucune image au reste du pipeline lorsqu'il y a une frame sur laquelle YOLO n'a détecté aucune feuille, ce qui représenterait le cas où une feuille est restée moins de 5

secondes sous la caméra, et donc ne pouvant être considérée comme devant être numérisée. Dans ce cas, il y a également de fortes chances que la feuille soit floue sur chacune des frames captées par la caméra.

Les paramètres pouvant être réglés sur cette partie sont :

- la résolution à laquelle on demande la caméra de fonctionner (dans la limite des résolutions existantes sur le périphérique utilisé)
- le seuil de confiance pour la détection d'objets
- les seuils de saturation et de valeur pour la couleur de l'image.
- le nombre de sous-blocs en lequel on divise l'image pour l'estimation du flou
- le nombre de sous-blocs de flou maximal à considérer pour calculer la moyenne des valeurs de flou
- le nombre limité d'images dans les buffers.

Le modèle de détection d'objets YOLOv11 atteint de bonnes performances en lui-même, ce qui est prometteur pour le pipeline : 0.8971 pour la justesse moyenne de classification des pixels. Mais son utilisabilité est subordonnée aux performances du modèle de segmentation objet/arrière-plan qui suit.

Rognage et correction de perspective

Afin d'obtenir les meilleures performances possibles pour la reconnaissance d'écriture manuscrite, il faut rogner au mieux l'image autour de la feuille et éviter que des éléments de l'arrière-plan viennent bruiter l'entrée du modèle de HTR.

Modèle basé sur la détection de contour

L'avantage du modèle reposant sur la détection de formes est qu'il fournit directement les quatre coins de la feuille. Ainsi, le changement de perspective se fait facilement par une transformation du quadrilatère en rectangle droit.

Modèle basé sur la détection par YOLO

L'image donnée en entrée de ce composant est la frame rognée autour de la bounding box donnée par YOLO. La segmentation feuille/arrière-plan se fait en plusieurs étapes :

- Conversion en niveaux de gris.

- Seuillage afin de récupérer les pixels les plus clairs de l'image.

- Transformations morphologiques (fermeture morphologique, i.e. dilatation puis érosion, puis ouverture morphologique, i.e. érosion puis dilatation) afin de "boucher les trous" du masque détecté, pour ne manquer aucun pixel (fermeture), puis afin de supprimer les points exceptionnels qui ne sont que du bruit sur notre détection (ouverture). On commence par la fermeture, car on préfère avoir un rappel aussi élevé que possible (et donc avoir plus de pixels de la feuille, quitte à avoir quelques pixels de l'arrière-plan).

- Enfin, on conserve la composante connexe la plus grande du masque obtenu, qui est assimilable à la feuille que l'on recherche, étant donné que la bounding box est censée être centrée autour de cette feuille.

- Cela nous donne le masque de la feuille cherchée. Afin d'effectuer la correction de perspective et de rotation, nous allons chercher les 4 coins de ce masque, que l'on va placer dans les 4 coins du rectangle de l'image envoyée à l'étape suivante du pipeline. Comme étape préliminaire, nous allons redresser la feuille, et chercher le rectangle d'aire minimale autour de ce masque, et transformer l'image pour placer ce rectangle à la verticale.

- Les 4 coins du masque sont choisis comme étant les points du masque les plus proches des 4 coins du rectangle d'aire minimale englobant le masque. Et on applique à la feuille la transformation qui positionne ces 4 coins du masque aux 4 coins de l'image que l'on va envoyer à la suite du pipeline. Cette transformation est effectuée à l'aide de la fonction *cv2.warpPerspective*.

Cette partie possède quelques paramètres qu'il est possible de régler :

- Le seuillage de l'image se fait avec un seuillage Otsu, qui calcule automatiquement le seuil pour lequel la variance de chacune des 2 classes (classe des pixels dont la valeur est inférieure au seuil et classe des pixels dont la valeur est supérieure au seuil). Cela permet d'avoir un peu plus de robustesse face aux différentes luminosités qui existent, mais on peut également choisir d'effectuer un seuillage classique, si la luminosité est suffisamment stable dans les conditions d'usage. Il faut alors définir la valeur du seuil.

- La taille du gabarit (*kernel*) utilisé pour les transformations morphologiques, qui est, ici, fixée à (5,5).

Comparaison de la feuille à l'historique des sauvegardes en BDD

L'étape suivante est la reconnaissance du texte manuscrit. Mais afin d'éviter de trop nombreux appels au modèle de HTR (Handwritten Text Recognition), nous avons implémenté une étape de comparaison entre la nouvelle image, que l'on souhaite ajouter à la BDD, et les dernières images qui ont été enregistrées en BDD.

Cette comparaison entre 2 images se fait uniquement avec de la vision, et ne se base sur aucune valeur sémantique des notes, et s'effectue à l'appel de la fonction *isSimilar(img1, img2)* dans le script `src/image_similarity/image_comparison.py`. Voici les différentes étapes de ce procédé :

- Conversion des 2 images en niveaux de gris.
- Alignement des images :

    - Détection et description des keypoints avec l'algorithme ORB.
    - Matching des keypoints sur la base de leurs descriptions.
    - Calcul de la transformation de perspective correspondante.

    - Transformation de la nouvelle image par la matrice calculée au-dessus.

- Seuillage de la feuille pour détecter les écritures sur la feuille.
- Inversion des niveaux de gris, afin que plus de poids soit donné aux écritures (noires) qu'au blanc de la feuille lors de l'interpolation suivante.
- Redimensionnement des images à une taille identique et plus petite (de manière à pixelliser l'image, et à n'avoir plus que l'information du positionnement des écritures). La taille cible utilisée ici est 40*40 pixels. Ce redimensionnement est effectué en utilisant une méthode d'interpolation particulière. La valeur d'un pixel de l'image de sortie est donnée par le calcul de la moyenne de Minkowski d'ordre 4 des valeurs des pixels correspondant à la même zone, de l'image d'entrée. Ceci correspond à l'idée de "grossir le trait du stylo" afin de pallier aux imprécisions de l'étape d'alignement.
- Calcul de la différence entre les deux images ainsi obtenues.
- Seuillage de cette différence afin de ne calculer que les valeurs de différence qui sont notables, et donc porteuses de sens.

Les paramètres de ce procédé sont nombreux, et il faut donc les régler pour l'adapter aux conditions d'utilisation :

- Nombre de keypoints calculés sur chaque image, ici réglé à 1000.
- Le taux de ces meilleurs keypoints qui sont conservés pour le calcul de la transformation, ici à 0.3.
- Le seuil utilisé pour le passage en binaire de l'image en niveaux de gris, ici à 140.
- l'ordre de la moyenne de Minkowski, ici à 4.
- la taille cible des images pour le redimensionnement par interpolation de Minkowski, ici à (40, 40).
- le seuil utilisé pour considérer un pixel de la différence entre 2 images pertinent, ici à 115.

Ainsi, s'il reste des pixels non nuls sur l'image en sortie de ce procédé, on considère que les 2 images ne sont pas identiques, et on peut donc transférer la nouvelle image au composant de transcription des notes manuscrites.

Le modèle de comparaison d'images ainsi implémenté ne permet pas de détecter la majorité des images identiques, mais a minima, il ne renvoie pas de faux positifs. C'est donc une voie intéressante à étudier pour éviter l'appel au HTR pour des doublons, mais la robustesse doit en être améliorée.

Transcription du texte

Une fois la feuille détectée, recadrée et redressée, le système applique un pipeline de reconnaissance d'écriture manuscrite visant à transformer le contenu papier en texte exploitable par le système Storylines. Cette étape constitue la passerelle critique entre le monde physique (notes écrites par le dispatcheur) et le monde numérique (base de données temporelles exploitées par l'interface Storylines).

Le pipeline comporte quatre étapes successives :

1) Reconnaissance manuscrite (OCR / Handwritten Text Recognition)

2) Filtrage de la qualité et normalisation du texte

3) Détection de versions et gestion des différences

4) Insertion dans la base et historisation

L'objectif est d'obtenir un texte fidèle, propre, stable dans le temps et versionné, tout en minimisant l'effort cognitif du dispatcheur (aucune action requise de sa part).

1. OCR : choix de la solution Teklia Ocelus

Après évaluation de plusieurs alternatives (OCR Mistral, fine-tuning TrOCR, modèle transformer à attention verticale), Teklia Ocelus a été retenu pour trois raisons principales :

- Robustesse en contexte réel sur l'écriture manuscrite française.
- Exposition d'informations structurées (texte + bounding boxes + score de confiance).
- API industrielle permettant un passage à l'échelle (latence, fiabilité, SLA).

Teklia renvoie chaque ligne manuscrite associée à un score de confiance. Ce score est filtré via le paramètre :

OCR_CONFIDENCE_THRESHOLD (par défaut : 0.5)

Interprétation :
- Si le score < 0.5 → la ligne est rejetée (trop incertaine).
- Si le score ≥ 0.5 → la ligne est conservée.

Recommandations d'ajustement :

- Notes propres, écriture régulière → seuil possible 0.6 à 0.7 (moins de bruit).
- Éclairage difficile, écriture cursive brusque → seuil 0.3 à 0.4 (moins de pertes).
- Phase de test / POC → 0.5 (compromis stabilité / rappel).

2. Normalisation du texte via LLM déterministe

Bien que l'OCR fournisse une transcription brute, la variabilité manuscrite impose une normalisation. Nous utilisons un modèle Mistral pour :

- corriger fautes mineures et variations d'orthographe
- uniformiser ponctuation, majuscules, format d'heure
- stabiliser la sortie (textes identiques => représentations identiques)

Paramètres clés :

NORMALIZATION_LLM_MODEL
(mistral-small-latest / mistral-medium-latest / mistral-large-latest)

NORMALIZATION_TEMPERATURE = 0.0 (obligatoire)
→ garantit une sortie déterministe.
Deux sorties identiques doivent toujours produire la même chaîne normalisée, sinon la détection de version deviendrait instable.

Guidelines :

- Volume important, coût limité → small
- Production avec exigence de qualité → medium
- Notes difficiles, budget confortable → large

3. Détection des nouveautés et gestion du versionnage

Une même feuille peut être captée plusieurs fois. Le système doit donc distinguer :

- feuille déjà vue sans modification → rien n'est stocké
- feuille déjà vue avec modifications → seules les lignes modifiées sont ajoutées
- nouvelle feuille → nouvel enregistrement

Mécanisme :
Comparaison du texte actuel avec les versions existantes via SequenceMatcher (distance de similarité) + règles métier.

Paramètres directs du module `add_data2db.py` :

NOTE_SIMILARITY_THRESHOLD (def. 0.7)
Si similarité ≥ 0.7 → considéré comme la même feuille
Ajustement :
- Écriture très stable → augmenter (0.8-0.9)
- OCR instable, variation forte → réduire (0.5-0.6)

ANTI_REPETITION_LENGTH_DIFF (def. 35)
Si différence de longueur > 35 caractères → on considère que ce n'est pas un doublon (évite calcul coûteux)
Ajustement :
- Notes longues avec ajouts successifs → augmenter (50-80)
- Notes courtes (2-4 lignes) → réduire (15-25)

ANTI_REPETITION_SIMILARITY_MIN (def. 0.1)
Seuil minimal pour bloquer doublons sur OCR instable
Cas rare mais critique : l'OCR peut "changer d'avis" sur la façon de lire la même image.
Ajustement :
- OCR très propre → monter (0.2-0.3)

- Éclairage changeant, reflets, écriture difficile → rester bas (0.05-0.1)

DIFF_MINOR_CHANGE_THRESHOLD (def. 0.90)
Si similarité > 90% → changement mineur ignoré
Ajustement :
- Besoin de granularité extrême (R&D, forensic) → baisser (0.8)
- Production avec tolérance aux micro-variations → monter (0.92-0.95)

Règle fondamentale :

On n'efface jamais une ligne.
Chaque modification crée une nouvelle version.
Historisation complète obligatoire pour lisibilité métier et auditabilité.

4. Résultat : texte exploitable, fiable et versionné

En sortie du pipeline OCR + normalisation + déduplication :

- Texte fidèle à la note papier
- Version unique par feuille physique, avec journal d'évolution
- Éviction automatique des doublons OCR
- Structure prête pour NER, regroupement temporel et Storylines

Comportement attendu côté interface :
- si aucune nouveauté détectée → aucune nouvelle carte
- si de nouvelles lignes sont ajoutées → nouvelle carte contenant uniquement les changements

Ce mécanisme garantit un flux propre et maîtrisé, tout en respectant l'usage naturel des dispatcheurs (écrire sur papier sans interaction logicielle supplémentaire).

Synthèse :

Si trop de doublons →
- diminuer *NOTE_SIMILARITY_THRESHOLD* (0.6)
- diminuer *ANTI_REPETITION_SIMILARITY_MIN* (0.05)

Si des modifications ne remontent pas →
- diminuer *DIFF_MINOR_CHANGE_THRESHOLD* (0.85)

Si trop de bruit OCR →
- augmenter *OCR_CONFIDENCE_THRESHOLD* (0.6)

Si perte de contenu utile →
- baisser *OCR_CONFIDENCE_THRESHOLD* (0.4)

Si coûts API élevés →

- passer *NORMALIZATION_LLM_MODEL* à small

Canal audio

Dans ce système, l'utilisateur est écouté en permanence, et un échantillon audio est enregistré automatiquement toutes les *n* minutes. Chaque segment audio est ensuite traité rigoureusement pour garantir une transcription de haute qualité avec un contexte cohérent.

Segmentation de la parole

Pour simplifier le travail du modèle de transcription, les enregistrements sont d'abord passés par un modèle PyAnnote de segmentation de type VAD (Voice Activity Detection). Ce modèle identifie les régions contenant de la parole humaine dans l'audio, permettant d'exclure le bruit ou les silences prolongés.

Deux paramètres principaux peuvent être ajustés pour affiner la détection :

- "min_duration_on" = Durée minimale d'un segment de parole à conserver (en secondes).
- "min_duration_off" = Durée minimale d'une pause pour qu'elle soit considérée comme une séparation (en secondes).

L'issue de cette étape est une liste de segments horodatés correspondant aux intervalles de parole détectés.

Il est possible de modifier les paramètres "min_duration_on","min_duration_off", ainsi que la durée de chaque segment audio en secondes directement dans le fichier `src/audio/pipeline_watcher.py`.

Transcription audio

Comparaison des modèles de transcription

Chaque segment identifié est ensuite transmis à un modèle de transcription, Nous avons comparé deux modèles de transcription automatique : Whisper (large-v3-turbo) et Wav2Vec2-large-fr. Le modèle Wav2Vec2 est spécifiquement entraîné sur de nombreux corpus français spécialisés dans ce domaine.

Pour comparer ces modèles, nous avons utilisé les métriques suivantes :

- Wer (Word Error Rate) : Le WER mesure le taux d'erreurs sur les mots transcrits et se calcule avec la formule :

$$\mathrm{WER} = \frac{S + D + I}{N}$$

où S est le nombre de substitutions, D le nombre de suppressions, I le nombre d'insertion, et N le nombre total de mots dans la référence.

- CER (Character Error Rate) : Le CER mesure les erreurs au niveau des caractères et se calcule de manière analogue :

$$\mathrm{CER} = \frac{S + D + I}{N}$$

où les variables sont les mêmes mais appliquées aux caractères.

- Temps de transcription : Le temps nécessaire pour transcrire un segment audio, mesuré en secondes par seconde d'audio.

Nous avons testé les modèles sur un segment du corpus Common Voice, sorti le 25/06/2025, qui contient de nombreux accents et des utilisateurs du monde entier lisant des textes issus de Wikipedia. La dernière mise à jour des deux modèles date de 2024, aucun des deux modèles n'a donc été entraîné sur notre jeu de test. Pour obtenir un jeu de test plus naturel, nous avons également utilisé des podcasts français issus de Youtube.

Data	Modèle	WER(%)	CER(%)	Durée moyenne
Common Voice	Whisper (large-v3-turbo)	11.14	5.05	0.37s
	Wav2Vec2 (large-fr)	9.69	3.61	**0.04s**
Poscast Youtube	Whisper (large-v3-turbo)	**7.95**	**3.60**	5min
	Wav2Vec2 (large-fr)	16.12	6.96	1min

TABLE 1 – Comparaison des modèles de transcription

Pour le corpus Common Voice, Wav2Vec2-large-fr est extrêmement rapide et légèrement plus précis que Whisper. Cela s'explique par le fait que ce modèle a été entraîné sur l'ensemble du corpus Common Voice et est donc habitué au vocabulaire de Wikipedia, mais cet avantage se réduit sur des données plus naturelles.
En effet, sur les podcasts YouTube, les résultats sont inversés, Whisper surpasse clairement Wav2Vec2-large-fr sur des contenus plus naturels, moins formalisés, et contenant des variations d'accent et de style de parole.

Après analyse des résultats, nous avons décidé de choisir le modèle Whisper pour la suite du processus. Chaque segment identifié est ensuite transmis à Whisper (large-v3-turbo) pour obtenir une transcription brute. Ce choix garantit que le texte reflète fidèlement le contenu oral tout en minimisant les erreurs liées aux silences ou aux bruits de fond. Ainsi, Whisper est préféré dans des contextes où l'audio est plus naturel ou varié, comme les podcasts, tandis que Wav2Vec2-large-fr reste plus adapté à des corpus standardisés et structurés comme Common Voice.

Transcription : Whisper

Une fois les segments de parole extraits, chacun est transmis au modèle Whisper (large-v3-turbo) pour la transcription. Whisper est un modèle multilingue de pointe conçu pour traiter des conditions variées d'enregistrement, incluant les environnements bruyants, les accents divers et les variations de débit de parole. Sa robustesse permet d'obtenir une

transcription fidèle même lorsque l'audio n'est pas optimal (bruits de fond, micro de faible qualité, intonations non standards, etc.).

Le fonctionnement est simple : le modèle écoute le segment audio, puis écrit ce qu'il entend sous forme de texte. Cela permet d'obtenir une transcription claire et fidèle, même si la personne parle vite ou si le son n'est pas parfait.

Pour chaque segment, Whisper fournit :

- la transcription du texte,
- un niveau de confiance,
- des informations temporelles permettant d'aligner le texte avec l'audio.

Ces données permettent ensuite d'organiser et d'améliorer le texte final si nécessaire (ponctuation, correction légère, etc.). Ce choix garantit donc une transcription fiable et adaptée à des situations de parole naturelle, comme des conversations ou des podcasts.

Exemple de transcription

Pour illustrer le fonctionnement du système, prenons un exemple où deux segments audio sont transcrits successivement :

Audio attendu :

- (15h30) : Appeler le COSE-P demain matin.
- (16h00) : Ligne 225 kV Charles–Trappes coupée jusqu'à 17h.

Sortie brute Whisper :

- (15h30) : Appeler le cos(p) demain matin.
- (16h00) : Ligne 225 kilovolts charle trappe coupée jusqu'à 17 heures.

Dans cet exemple, le problème principal provient du fait que Whisper est sensible aux abréviations, aux noms propres et aux noms de villes, particulièrement fréquents dans les communications terrain (des dispatcheurs, opérateurs, techniciens, etc.). Sans contextualisation, ces termes peuvent être mal reconstruits ou interprétés de manière erronée par le modèle.

Dans le fichier `src/audio/whisper_transcriber.py`, le modèle Whisper utilisé peut être modifié parmi les options suivantes :

- 'tiny', 'base', 'small', 'medium',
- 'large-v1', 'large-v2', 'large-v3', 'large'
- 'large-v3-turbo', 'turbo'

Post-traitement

Le problème principal réside dans le fait que Whisper est très sensible aux abréviations utilisées par les dispatcheurs, ainsi qu'aux noms propres et aux noms de villes. C'est

pourquoi nous faisons appel à un LLM (Mistral-large-latest) chargé de contextualiser et restructurer la transcription. Ce modèle permet notamment de corriger :

- Noms propres
- Abréviations
- Noms de villes / sites

Afin de renforcer sa fiabilité, un dictionnaire spécialisé lui est fourni pour guider la reconnaissance et la normalisation des entités clés. Ces éléments sont entièrement personnalisables dans le dossier `src/audio/dictionary`.

Sortie finale corrigée :

- (15h30) : Appeler le COSE-P demain matin.
- (16h00) : Ligne 225 kV Charles-Trappes coupée jusqu'à 17h00.

Ainsi, la combinaison Whisper + LLM + dictionnaire métier assure une transcription robuste, contextualisée et adaptée au domaine opérationnel.

Test pipeline

Pour tester tout le pipeline audio, il suffit de se rendre dans `src/audio/pipeline_watcher`.
Un dossier test sera créé pour stocker les audios bruts. Ils seront ensuite segmentés puis envoyés dans le dossier tmp, transcrits et enfin nettoyés.

Vous pouvez vérifier les résultats directement dans le fichier :
`src/audio/tmp/transcriptions_log.json`

Structuration des informations textuelles

Reconnaissance d'entités

Afin de permettre un suivi des évènements et donc de s'inscrire dans le projet Storylines, un système de reconnaissance d'entités a été implémenté. Que ce soit par la vidéo ou par l'audio, nous obtenons un texte "nettoyé" qui retranscrit l'information capturée. Il est alors intéressant d'y extraire certains éléments pouvant identifier l'évènement dans lequel s'inscrit la note. Les catégories que nous avons retenues sont les suivantes:

Catégories	Exemples
GEO	-noms de lieux, villes, régions.
ACTOR	-personnes, entreprises, équipes.
DATETIME	-dates, horaires, marqueurs temporels.
EVENT	-évènements spécifiques (incidents, manœuvres, interventions...).
INFRASTRUCTURE	-éléments du réseau (ouvrages électriques, lignes, postes...).
OPERATING_CONTEXT	-informations variées sur les opérations réalisées.
PHONE_NUMBER	-numéro de téléphone.
ELECTRICAL_VALUE	-valeurs électriques (tension et puissance).
ABBREVIATION_UNKNOWN	-abréviations inconnues mais certainement porteuses d'informations.

TABLE 2 – Répartition des échantillons par catégorie

Pour reconnaître les mots correspondant à ces catégories d'entités, le texte nettoyé passe d'abord par une phase de traduction. L'idée ici est de reconnaître la présence d'abréviations techniques connues et de les remplacer par leur traduction complète afin de faciliter le travail ultérieur.

Concrètement, une phrase comme "MNV à réaliser à 14h" devient "Manoeuvre à réaliser à 14h". Ce texte traduit est ensuite envoyé à un modèle de langage Mistral via un appel API qui joue le rôle d'extracteur d'entités. A l'aide d'un prompt détaillé, ce dernier est capable d'identifier les différentes entités présentes dans le texte et de les retourner sous forme d'un dictionnaire catégories - entités correspondantes. Ainsi, le contenu issu de la numérisation de la note désormais accompagné de ses entités extraites est prêt à être envoyé à la base de données.

Lorsqu'une nouvelle entrée est envoyée à la base de données, elle n'est pas encore associée à un événement. Afin de le déterminer, cette nouvelle entrée va être comparée une à une à celles en base en utilisant les entités extraites.

La comparaison repose sur la similarité des entités catégorisées, pondérées selon l'importance de chaque catégorie. Concrètement, pour chaque catégorie d'entité en commun, on vérifie si les entités extraites des deux textes sont suffisamment proches. Chaque catégorie a un poids qui reflète son importance dans l'identification d'un événement. On calcule alors un score global pondéré, qui correspond à la somme des correspondances observées dans chaque catégorie divisée par la longueur de la plus petite des deux listes de la catégorie. On multiplie ensuite chaque somme par le poids de la catégorie, puis on les somme entre elles. Ce score représente donc la proportion des entités similaires pondérées par leur importance selon leur catégorie entre les deux textes.

Pour décider si deux textes décrivent le même évènement, ce score global est comparé à un seuil qui est une proportion du poids total des catégories communes. Si ce score dépasse ce seuil, cela signifie que les deux textes partagent suffisamment d'éléments en commun et qu'ils sont très probablement liés au même évènement. Dès qu'un élément de la base est jugé être en lien avec notre nouvelle entrée, son identifiant d'évènement va être

celui retenu pour la nouvelle note. Si aucune ancienne note ne peut être reliée à notre entrée, un nouvel identifiant d'évènement incrémenté lui sera alors attribué.

On peut bien évidemment jouer sur la sensibilité du seuil de similarité (SIMILARITY_THRESHOLD) pour rendre le modèle plus ou moins restrictif dans sa comparaison d'entités mais aussi sur la proportion minimale du poids total (PROPORTION) qu'il faut pour établir que deux textes correspondent au même évènement.

Synthèse temporelle des informations

Après avoir obtenu les informations provenant de sources manuscrites et audio, nous procédons à leur organisation afin de pouvoir les analyser efficacement. Chaque information est d'abord soigneusement relevée et horodatée pour garantir une chronologie précise. Cette étape permet de disposer d'une base solide pour la suite du traitement.

Organisation en blocs temporels

Les informations sont ensuite regroupées en blocs temporels. Chaque bloc correspond à un intervalle de temps où les informations sont suffisamment proches les unes des autres pour être considérées comme liées. La création de ces blocs repose sur deux paramètres essentiels :

- MAX_GROUP_DURATION : chaque bloc ne peut pas dépasser une certaine durée totale. Si l'ensemble des informations d'un bloc dépasse cette limite, un nouveau bloc est automatiquement créé. Cela permet de garantir que chaque bloc reste concis et focalisé sur un intervalle temporel restreint, évitant ainsi l'accumulation d'informations trop dispersées dans le temps.
- MAX_PAUSE : il s'agit du temps maximal autorisé entre deux informations pour qu'elles appartiennent au même bloc. Si l'écart entre deux informations dépasse ce seuil, un nouveau bloc est créé. Ce paramètre permet de segmenter les informations selon leur proximité réelle dans le temps, ce qui améliore la cohérence et la pertinence de chaque bloc.

Grâce à ces deux paramètres, les données sont organisées de manière structurée et lisible. Chaque bloc reflète une période homogène d'informations, facilitant la synthèse et l'analyse chronologique.

Ces hyperparamètres peuvent être modifiés dans le fichier `src/frontend/pages/config.py`.

Génération de synthèses par bloc

Une fois les blocs temporels constitués, chaque bloc fait l'objet d'une synthèse automatique. Cette synthèse résume les points essentiels de l'ensemble des informations contenues dans le bloc, tout en respectant l'ordre chronologique. Le résultat est une lecture claire et structurée des événements ou informations collectées.

Présentation chronologique

Les synthèses sont ensuite présentées sous forme de cartes chronologiques. Chaque carte correspond à un bloc temporel et contient sa synthèse ainsi que les dates et heures de début et de fin du bloc. Ce format visuel facilite la lecture et permet de naviguer rapidement à travers les différentes périodes d'informations.

Conclusion

Le prototype D-ScrAibe démontre la faisabilité d'un système capable de numériser et structurer automatiquement les notes manuscrites et les échanges oraux en salle de dispatching. En combinant captation vidéo, reconnaissance d'écriture, transcription audio et traitement du langage naturel, il permet d'alimenter l'outil Storylines sans perturber le travail des opérateurs.