YNOV informatique Toulouse

Titre RNCP 39586

Ingénieur en science des données spécialisé en infrastructure data ou en apprentissage automatique.

Rapport bloc 1

COLLECTER, TRANSFORMER ET SECURISER DES DONNEES

Collecter, transformer et sécuriser des données pour tester la vulnérabilité des attaques par canaux auxiliaires (SCA) sur un trafic radiofréquence chiffré.

Rédigé par	*Responsabilité*
Maxence colvray	Apprenti datascientist

Table des matières

1 Description synthétique du projet (C1.1.1) 4
1.1 Qu'est-ce qu'une attaque par canaux auxiliaires ? 4
1.2 Objectifs opérationnels 4
1.3 Contexte technique 4
2 Stratégie de collecte (C1.1.1) 5
2.1 Objectif technique du pipeline de données en temps réel. 5
2.2 Source de données 5
2.3 Données utiles pour répondre à la problématique 6
2.4 Résumé des moyens envisagés 6
3 Exemple de collecte de données (C1.1.2) 7
3.1 Script du client API via un socket TCP 7
3.2 Exhaustivité des données collectées 9
3.3 Exactitude des données collectées 9
3.4 Cadre réglementaire 9
4 Méthode d'automatisation (C1.1.3) 10
4.1 Planning d'ordonnancement 10
4.2 Script principal d'ordonnancement et déclenchement de la collecte 10
4.3 La fonction signal_collect 11
4.4 Automatisation de la collecte et boucle de contrôle horaire 11
5 Stratégie de stockage (C1.2.1) 12
5.1 Définition de l'architecture de stockage Hydrique 12
5.2 Modèle de données 12
6 Choix des solutions de stockage des données (C1.2.2) 13
6.1 Schéma et choix des technologies de stockage 13
6.2 Extraits de la base documentaire centrale 14
6.3 Extraits de la base de métadonnées de séries temporelles (C1.2.2 & C1.3.2) 15
7 Traitement de données (C1.3.1) 16
7.1 Langage et pipeline de traitement de données. 16
7.2 Choix des librairies utilisés 16
8 Présentation des données transformées (C1.3.2) 17
8.1 Transformation signal audio au format WAV 17
8.2 Transformation signal audio au format WAV chiffré 17
8.3 Spectrogramme et jointure temps-fréquence. 18
9 Talend pour l'ingestion et la transformation de données (C1.3.3) 19
9.1 Job_Mongo_Ingest métadonnées dans MongoDB 19
9.2 Job_Influx_Write métadonnées de séries temporelles vers InfluxDB 3 19

9.3 Job_Transform spectrogramme & features via Python 20
10 Politique de sécurité (C1.4.1) 21
11 Conclusion 22

Terminologie

SCA	Side Channel Attack ou attaque par canaux auxiliaire
EM	Electromagnétique
Radiofréquence	La radiofréquence est la gamme d'ondes électromagnétiques (3khz à 300Ghz) utilisée pour les transmissions sans fil
Simple-Scrambling	Méthode pour chiffrer un signal radio
Données I/Q	Représentation numérique brute du signal radio
SDR	(Software Defined Radio)
ZCR	(Zero Crossing Rate) fréquence à laquelle le signal change de signe (passe par zéro)
RMS	(Root Mean Square) mesure de l'énergie moyenne d'un signal, liée à sa puissance perçue
Fréquence fondamentale	fréquence la plus basse d'un signal périodique, perçue comme sa hauteur
Centroïde spectral	mesure qui indique où se situe le centre de gravité du spectre, liée à la brillance d'un son.

1 Description synthétique du projet (C1.1.1)

1.1 Qu'est-ce qu'une attaque par canaux auxiliaires ?

Une attaque par canaux auxiliaires (ou SCA, pour Side Channel Attack) est une méthode de piratage pour laquelle on ne rentre pas dans le système, mais où l'on exploite les fuites physiques indirectes (comme les ondes électromagnétiques, la consommation électrique ou le temps d'exécution) pour en déduire des informations confidentielles. Ce projet concerne les SCA appliquées aux fuites d'ondes électromagnétiques, appelé aussi SCA EM. On retrouve 2 cas d'usage principaux :

Cryptanalyse :

Déchiffrage de clé de chiffrement (ex : AES, RSA, NTRU) en classifier des traces EM généré par le processeur pendant les opérations de déchiffrement.

Classification d'un trafic radiofréquence chiffré :

Analyser un signal radio chiffré, provenant par exemple d'une borne wifi, d'un satellite, d'une radio, pour en déduire des informations sur l'activité en cours. C'est-à-dire que même sans déchiffrer le contenu, on peut analyser des paternes permettant de classifier certaines caractéristiques du signal (ex : provenance, matériel utilisé, type de contenus, identité du locuteur, etc.).

Plusieurs contre-mesures existent et ont besoin d'être testées pour palier à ce type d'attaque. Parmi les principales méthodes de chiffrage appliquées directement au signal, on retrouve par exemple l'inversion de spectre, le saut de fréquence ou le Simple-Scrambling.

1.2 Objectifs opérationnels

Dans ce projet, nous allons créer un jeu de données, qui servira par la suite, à entrainer des modèles et tester des attaques sur un trafic chiffré. Les méthodes pour faire cela sont aussi utilisées pour la cryptanalyse, car elles exploitent les mêmes principes d'observation (collecte de signaux électromagnétique) et d'attaques (classification par apprentissage automatique).

L'objectif opérationnel est donc de préparer un jeu de données (avec des données audios ici collectées sur la bande FM) pour, plus tard dans le cadre d'un autre projet, pouvoir lancer des attaques basées sur l'apprentissage automatique afin d'évaluer la vulnérabilité d'un flux radiofréquence (par exemple une radio chiffrée).

1.3 Contexte technique

Pour des raisons matérielles et pédagogiques, le jeu de données utilisé provient d'un **flux RF capté en direct** sur la bande FM, sous forme de signal radio puis audio. Ce projet vise, par exemple, à simuler des attaques sur une radio professionnelle afin d'évaluer la résistance de son chiffrement face à des techniques d'analyse par canaux auxiliaires.

Les signaux seront collectés en direct, démodulés, puis stockés de manière sécurisée en base de données. Une fois un chiffrement simple appliqué (par exemple, simple scrambling), les données pourront être labellisées et utilisées pour entraîner un modèle. Cela permettra ensuite d'estimer, à partir du signal chiffré, des caractéristiques comme le genre du locuteur, la langue parlée ou même une identité probable.

Ce travail s'inscrit donc dans un projet plus large couvrant l'ensemble des blocs de compétences du titre RNCP, en lien avec la collecte, la transformation, la sécurisation et l'analyse de données. **L'objectif final ici est de disposer d'un jeu de données exploitable pour entrainer des modèles et mener plus tard des attaques sur un trafic chiffré** provenant d'une radio de communication professionnelle.

2 Stratégie de collecte (C1.1.1)

2.1 Objectif technique du pipeline de données en temps réel.

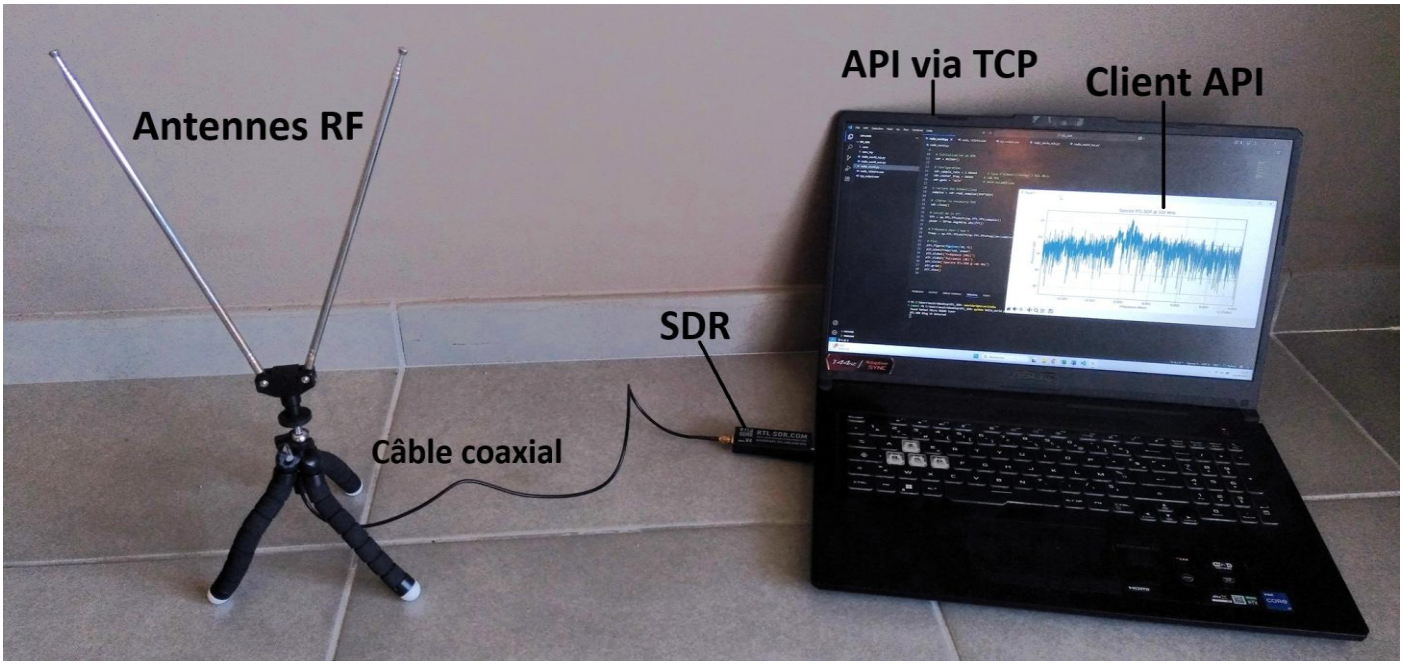
Collecter en continu des signaux RF :
Le pipeline présenté ici, vise à collecter un signal RF en temps réel, accessible depuis une API via un port TCP pour capter des signaux radio dans la bande FM simulant des communications radio humaines.

Réception et démodulation du signal :
Définir une largeur de bande appropriée, extraire le contenu brut des signaux via une antenne et démoduler les signaux reçus via l'antenne.

Transformer, étiqueter et stocker les données audios :
Appliquer un chiffrement simple (comme Simple Scrambling) sur l'audio récupéré, ajouter des étiquettes utiles (par exemple : langue, genre, locuteur, méthode de chiffrement), puis enregistrer les données de façon organisée pour pouvoir les utiliser plus tard pour entrainer et tester des modèles d'attaques par canaux auxiliaires sur une radio chiffrée.

2.2 Source de données

La source de données utilisée est une chaîne d'acquisition RF composée d'une antenne, d'un câble coaxial, d'un SDR (Software Define Radio) chargé de numériser le signal. Ensuite les données numériques sont diffusées sur un serveur TCP, accessible depuis un port dédié, permettant à un client API d'exploiter le flux de données en temps réel.



À ce stade, les données provenant de l'API ne sont pas encore directement exploitables. Il s'agit d'un flux de données I/Q (In-phase & Quadrature), une représentation numérique brute du signal radio. Ce signal devra ensuite être filtré et démodulé (ici en FM) afin d'être converti en signal audio exploitable.

2.3 Données utiles pour répondre à la problématique

Les données nécessaires proviennent du flux I/Q (In-phase & Quadrature) transmis en continu via une API. Ce flux sera filtré et démodulé en FM afin d'extraire le signal audio. A la suite de cela on aura besoin de transformer les données numériques sous plusieurs formats pour répondre à la problématique :

Séries temporelles classique (time series) :
Le signal démodulé peut être conservé sous forme de série temporelle, c'est-à-dire une séquence numérique représentant simplement l'évolution de l'amplitude du signal audio dans le temps. Ce format est adapté à l'analyse de signal numérique pour des plages de fréquences relativement basses (comme la radio FM ou les radios de communication terrestres). On pourra aussi transformer le signal en plusieurs caractéristiques fréquentielles sous forme de séries temporelles qui seront utilisées comme des métadonnées associées au signal audio.

Images de spectre (spectrogrammes) :
Le signal audio peut être converti en spectrogramme, une image représentant l'évolution des fréquences dans le temps. Ce format est obtenu par une transformée de Fourier et permet de visualiser les composantes fréquentielles du signal. Il est particulièrement adapté à l'analyse automatique par des modèles de vision, comme certains utilisés en deep learning pour faire de la classification.

Fichier audio .WAV :
Le signal peut être exporté au format WAV, un format audio non compressé qui préserve la qualité d'origine. Il permet une écoute directe, une annotation manuelle (genre, langue, locuteur) et pourra servir ultérieurement à labelliser les données, à travers simplement l'écoute humaine, pour l'entraînement de modèles supervisés. Il y aura également un deuxième fichier audio chiffré c'est ce dernier qui sera la source de données utilisée pour lancer des attaques dans un futur projet.

2.4 Résumé des moyens envisagés

Pour mettre en place le pipeline de collecte, plusieurs moyens matériels et logiciels sont mobilisés :

- Une chaine d'acquisition RF basé sur un SDR (Software Defined Radio) relié à une antenne qui capte les signaux radio, qui seront numérisés et transmis en temps réel via TCP.
- Un client logiciel qui se connecte à ce flux pour filtrer, démoduler (en FM) et transformer le signal en différents formats exploitables : séries temporelles, spectrogrammes, et fichiers audio WAV.
- Les outils utilisés incluent notamment rtl_tcp.exe, des scripts Python, et des bibliothèques de traitement du signal (ex : NumPy, SciPy.signal, rtlsdr, socket, Librosa).
- Les données ainsi produites seront, dans un futur projet, labellisées pour entraîner et tester des modèles d'apprentissage dans le cadre d'attaques par canaux auxiliaires.
- Pour constituer un dataset exploitable, il est prévu de collecter au minimum 1000 extraits de signaux FM d'une durée de 5 à 10 secondes chacun, afin d'assurer une diversité suffisante pour l'entraînement et l'évaluation des modèles.

3 Exemple de collecte de données (C1.1.2)

3.1 Script du client API via un socket TCP

Le client API mis en place permet de parcourir l'ensemble des étapes nécessaires à la collecte et au traitement du signal RF. Après la connexion au serveur rtl_tcp, le client configure le SDR à travers des commandes (fréquence, taux d'échantillonnage, gain), puis lit en temps réel le flux binaire d'échantillons I/Q transmis. Ces données sont ensuite filtrées et démodulées en FM puis converties en signal audio pour être enregistrées au format .WAV dans cet exemple.

Importation des modules nécessaires :

```
import socket
import struct
import numpy as np
import scipy.signal as signal
import soundfile as sf
```

Socket : gère la connexion TCP avec le serveur SDR (rtl_tcp.exe)
Struct : formate les commandes binaires envoyées au SDR
Scipy.signal : effectue le traitement du signal (filtres, FM, resampling)
Numpy : pour manipuler les données
Soundfile : sauvegarder le signal audio final au format WAV

Connexion au serveur TCP :

```
# Connexion au serveur rtl_tcp
# print(" Connexion à rtl_tcp ...)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('127.0.0.1', 1234))
```

Envoie des commandes binaires avant de lancer l'enregistrement en temps réel :
Ces commandes sont envoyées au SDR et permettent de configurer la fréquence, le taux d'échantillonnage et le gain.

```
#  Configuration SDR (protocole rtl_tcp)
def send_cmd(cmd, param):
    buf = struct.pack('>BI', cmd, param)
    sock.sendall(buf)

send_cmd(0x01, CENTER_FREQ)    # SET_FREQUENCY
send_cmd(0x02, SAMPLE_RATE)    # SET_SAMPLE_RATE
send_cmd(0x03, GAIN_DB10)      # SET_TUNER_GAIN
```

Connexion au serveur TCP :
Ici on calcule combien d'échantillons I/Q il faut lire pour la durée choisie (ici 2 secondes), puis on lit les données brutes en temps réel jusqu'à atteindre la taille attendue.

```
# Lecture des échantillons IQ
num_samples = int(SAMPLE_RATE * DURATION)
num_bytes = num_samples * 2  # I + Q

print(f" Lecture de {num_samples} échantillons IQ...")
iq_raw = b''
while len(iq_raw) < num_bytes:
    chunk = sock.recv(num_bytes - len(iq_raw))
    if not chunk:
        raise IOError("Connexion perdue")
    iq_raw += chunk

sock.close()
print("Lecture terminée.")
```

Traitement du signal I/Q

Ici on effectue plusieurs opérations de traitement de signal qui ne seront pas détaillées dans ce rapport. C'est à ce moment-là que l'on démodule le signal, c'est-à-dire que l'on convertie le signal brut en signal FM compréhensible par un humain.

```
#  Conversion IQ brut en complexe
iq = np.frombuffer(iq_raw, dtype=np.uint8).astype(np.float32)
iq = (iq - 127.5) / 127.5
iq = iq[::2] + 1j * iq[1::2]

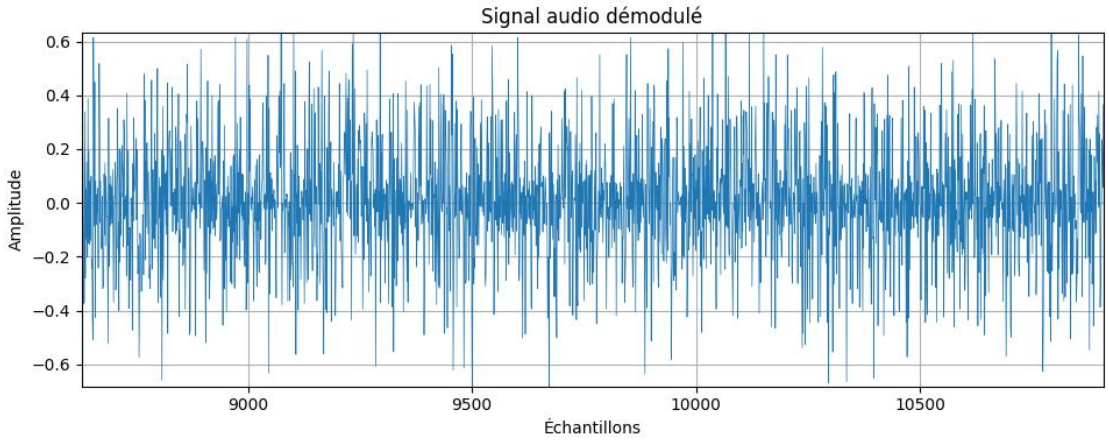
#  Filtrage passe-bas
taps = signal.firwin(101, 100e3 / (SAMPLE_RATE / 2))
iq_filtered = signal.lfilter(taps, 1.0, iq)

#  Démodulation FM
angle = np.angle(iq_filtered)
demod = np.diff(np.unwrap(angle))

#  Filtrage audio
b, a = signal.butter(5, AUDIO_RATE / (SAMPLE_RATE / 2))
audio_filtered = signal.lfilter(b, a, demod)
```

Affichage du signal et enregistrement au format .WAV :

Le signal audio démodulé est affiché sous forme de série temporelle, ici avec la librairie *matplotlib*, pour visualiser son évolution dans le temps. Il est ensuite enregistré au format WAV pour permettre l'écoute ou une analyse ultérieure. Ici on a l'amplitude en ordonnée et les numéro d'échantillon en abscisses.



Pour résumer le script Python développé ici agit comme un client API qui vient se connecter au serveur *rtl_tcp*, qui expose une interface bas niveau pour un récepteur SDR. Le client API ici permet de configurer la fréquence, le gain et le taux d'échantillonnage, puis de recevoir en temps réel un flux de données I/Q brutes. Ces données sont ensuite traitées localement, un filtrage RF est appliqué, suivi d'une démodulation FM, d'un filtrage audio, puis d'une conversion en signal audio au format standard WAV.

Ce processus offre plusieurs avantages, une collecte automatisable, un contrôle direct sur le SDR, et une haute-fidélité du signal capté. Toutefois, certaines limites subsistent car le protocole utilisé est propriétaire et non structuré, et le flux ne contient aucune métadonnée, ce qui impose de porter une attention particulière à l'exhaustivité et à l'exactitude des données collectée.

3.2 Exhaustivité des données collectées

Les premiers tests d'attaques par apprentissage automatique porteront sur des tâches de classification du genre et d'identification de locuteurs à partir des signaux audios démodulés.
Dans ce cadre, il est essentiel de disposer d'un jeu de données diversifié, incluant du bruit et des voix de genres et d'identités des locuteurs différentes, afin de garantir une exhaustivité suffisante pour, plus tard, entraîner et évaluer les modèles de manière fiable.

Pour ce faire, il sera nécessaire de cibler du contenu radio provenant de plusieurs émissions, diffusées à différents moments et sur différentes fréquences. Cela permettra de capturer une variété de locuteurs, de contextes de parole et de types de contenus (interviews, chroniques, débats, etc.). Cette diversité est indispensable pour constituer un jeu de données représentatif, adapté aux premières expérimentations en classification.

Il sera également possible de labelliser automatiquement le genre des locuteurs en utilisant des modèles déjà entraînés disponibles dans la librairie PyAudioAnalysis, ce qui permettra de faciliter l'analyse statistique, comme la répartition homme/femme dans les enregistrements collectés.

3.3 Exactitude des données collectées

L'exactitude des données collectées est un critère essentiel pour garantir la fiabilité des résultats obtenus lors des futures phases d'analyse ou d'entraînement des modèles. Dans ce projet, elle repose sur la capacité à capturer un signal audio clair et fidèle à partir du flux I/Q reçu. Cela suppose une configuration rigoureuse du SDR (fréquence centrale, gain, taux d'échantillonnage), ainsi qu'un traitement du signal précis (filtrage, démodulation FM, resampling).

En Python, il est possible de calculer automatiquement le ZCR (Zero Crossing Rate) pour détecter si le signal audio est exploitable ou s'il s'agit de bruit, voire d'un signal corrompu ou vide à cause d'un problème de réception.

3.4 Cadre réglementaire

Ce projet repose sur la réception passive de signaux radio via une antenne connectée à un récepteur SDR, sans émission. À ce titre, il respecte la réglementation française sur l'écoute des bandes hertziennes. En particulier, la réception de signaux dans la bande FM (88–108 MHz) est permise sans autorisation, car il s'agit d'un spectre public et non protégé.
Cependant, le traitement et l'analyse de ces signaux doivent être encadrés pour respecter le droit d'auteur et la protection des données personnelles.

En ce qui concerne l'analyse de trafic chiffré, il est important de rappeler que l'antenne et le SDR ne doivent en aucun cas être utilisés pour intercepter des communications privées, protégées ou cryptées (par exemple : services de secours, liaisons professionnelles, télécommunications militaires). Cette pratique est strictement interdite par l'article L33-3 du Code des postes et des communications électroniques, qui proscrit l'écoute des correspondances non destinées au public. C'est entre autres la raison pour laquelle on travaille ici sur des données de la bande FM.

4 Méthode d'automatisation (C1.1.3)

4.1 Planning d'ordonnancement

Un ordonnanceur développé en Python permet d'automatiser la collecte des données RF en suivant un planning défini dans un fichier Excel. Ce fichier contient, pour chaque session, la date, l'heure de début, l'heure de fin et la fréquence à surveiller. Le script lit ce tableau grâce à la bibliothèque pandas, puis compare en continu l'heure système aux créneaux définis. Lorsqu'un créneau actif est détecté, la fréquence correspondante est transmise à la fonction de capture, déclenchant automatiquement l'enregistrement du signal. Ce mécanisme repose sur une boucle de surveillance et sur la conversion des horaires en objets datetime afin d'assurer une comparaison fiable. L'ordonnanceur garantit ainsi une acquisition ciblée et ponctuelle des données, tout en supprimant la nécessité d'un déclenchement manuel et en assurant la reproductibilité du processus.

Date	Start Time	End Time	Frequency (MHz)
2025-08-05	12:00	14:00	104.1
2025-08-07	10:00	11:00	104.1
2025-08-07	15:00	15:30	104.1
2025-08-07	16:00	16:15	102.3
2025-08-08	12:00	13:00	102.3
2025-08-08	10:00	10:30	102.3

4.2 Script principal d'ordonnancement et déclenchement de la collecte

Le fichier main.py constitue le script principal chargé d'orchestrer la collecte automatisée des données RF, il lit en continu le planning Excel, détecte les créneaux actifs et déclenche la capture sur la fréquence correspondante. Pour cela, il appelle la fonction signal_collect du module services, qui renvoie les données audios démodulées ainsi que leurs fréquences d'échantillonnage. Ces données sont ensuite exploitées dans *main.py*. On pourra ensuite les afficher ou les transformer en fichier WAV pour archivage et analyses ultérieures. Les utilitaires du répertoire */utils* (validation via ZCR et écriture de logs) sont utilisés depuis la fonction *signal_collect* présente dans le répertoire */services*, ce qui maintient une séparation claire entre l'ordonnancement, la logique métier de collecte et les fonctions techniques, rendant l'ensemble plus lisible, maintenable et évolutif.

```python
def main_loop():
    while True:
        try:
            df = pd.read_excel("config\planning.xlsx")
            now = datetime.now()
            now_str = now.strftime("%Y-%m-%d")
            now_time = now.time()

            for _, row in df.iterrows():
                date_str = str(row["Date"])[:10]
                if date_str == now_str:
                    if in_time_range(row["Start Time"], row["End Time"], now_time):
                        freq_mhz = float(row["Frequency (MHz)"])
                        print(f"Créneau actif trouvé → Lancement sur {freq_mhz} MHz")
                        audio, sr = signal_collect(freq_mhz)
```

4.3 La fonction signal_collect

La fonction signal_collect réalise la capture et le pré-traitement du signal FM, elle configure le SDR (taux d'échantillonnage, mode de gain manuel, gain du tuner, fréquence centrale passée en paramètre), lit les échantillons IQ bruts, les convertit en complexe, applique un filtre passe-bas RF, puis effectue la démodulation FM. Le signal résultant est ensuite filtré côté audio, re-échantillonné à 16 kHz et normalisé.

À l'issue de ce traitement de signal, comme on peut le voir ci-dessous on calcule automatiquement le Zero-Crossing Rate (ZCR) pour calculer la présence des signaux utile vs silence/bruit afin d'effectuer un contrôle qualité automatique. Ensuite en fonction du seuil, on décide si l'échantillon est exploitable ou non (acceptation/rejet), et cette décision peut être consignée dans les logs pour la traçabilité. On pourra alors pour terminer, retourner les données prêtes à l'usage sous la forme d'un tuple : le signal audio démodulé (tableau NumPy) avec la fréquence d'échantillonnage audio, pour que ces données puissent être exploitées depuis le fichier main.py.

```
# --- Filtrage audio (identique à ton script)
b, a = signal.butter(5, AUDIO_RATE / (SAMPLE_RATE / 2))
audio_filtered = signal.lfilter(b, a, demod)

# --- Resampling vers AUDIO_RATE * DURATION
resampled = signal.resample(audio_filtered, int(AUDIO_RATE * DURATION))

# --- Normalisation (protégée)
max_val = np.max(np.abs(resampled))
if max_val > 0:

    resampled = resampled / max_val
# --- Calcul ZCR + décision
zcr = compute_zcr(resampled)
keep = should_continue(zcr)

return resampled.astype(np.float32), AUDIO_RATE
```

4.4 Automatisation de la collecte et boucle de contrôle horaire

Pour résumer, l'automatisation s'appuie sur une boucle d'ordonnancement qui lit périodiquement le planning Excel, identifie les créneaux actifs, puis déclenche signal_collect dès qu'un créneau est en cours. À chaque itération, le script récupère les données audios, applique la validation ZCR, journalise l'état OK ou non OK et, si valide, permet l'affichage et l'export en WAV depuis le fichier main. Une fois la capture terminée, le programme reboucle immédiatement, s'il est toujours dans la même plage horaire il relance une nouvelle acquisition, sinon il repasse en veille et réévalue le planning à intervalle régulier (avec un *time sleep* de 60s), jusqu'au prochain créneau prévu. Cette logique assure une collecte régulière et continue pendant les fenêtres planifiées, minimise les interventions manuelles et évite les doublons tout en respectant strictement les horaires définis.

```
        now_time = now.time()

        for _, row in df.iterrows():
            date_str = str(row["Date"])[:10]
            if date_str == now_str:
                if in_time_range(row["Start Time"], row["End Time"], now_time):
                    freq_mhz = float(row["Frequency (MHz)"])
                    print(f"Créneau actif trouvé → Lancement sur {freq_mhz} MHz")
                    audio, sr = signal_collect(freq_mhz)
        else:
            print("Aucun créneau actif.")

    except Exception as e:
        print(f"Erreur dans main loop : {e}")

    time.sleep(60)  # attente 60s
```

5 Stratégie de stockage (C1.2.1)

5.1 Définition de l'architecture de stockage Hydrique

La stratégie de stockage repose sur une architecture répartie en 3 selon la nature et l'usage des données :

Les informations descriptives de chaque signal (fréquence, durée, lien vers les fichiers audio, métadonnées etc.) auront besoin d'être stockées sous forme de documents en utilisant un format de données non structuré ou semi-structuré. Cette base documentaire constituera le point d'entrée principal pour la recherche et l'exploration des données. Elle devra donc rendre l'indexation efficace, et offrir une flexibilité de schéma pour adapter dynamiquement les champs en fonction des besoins, comme l'ajout de labels, de métadonnées spécifiques, ou d'informations techniques supplémentaires selon la bande de fréquence utilisée ou le type de démodulation.

Les signaux audios collectés, traités et démodulé devront être enregistrés dans une base de fichier audio qui permet de garder l'intégralité des données sans compression ou perte de données. Il sera également nécessaire de pouvoir convertir les fichiers audios vers n'importe quel autre format de fichier. Troisièmement le format de fichier audio devra permettre de rendre les signaux facilement exploitables en python.

Les images de spectres Les images spectrales des signaux audios prévues dans le champ *output* représentent également une ressource importante, car elles pourront être exploitées en complément du signal audio.

Les métadonnées physiques (caractéristique temporelle, harmonique ou fréquentielle) de chaque signal audio devront être enregistré en tant que série temporelle. Ces caractéristiques permettront de jouer le rôle de métadonnées physique pour analyser un signal en suivant son intensité, sa hauteur et son timbre au fil du temps. Cela servira à comparer différents signaux et à explorer rapidement des jeux de données audio. Ces métadonnées pourront éventuellement être utilisées pour tester des modèles simples de Machine Learning afin réaliser des tâches de classification simple (ex : détection de genre sur des fichiers audios chiffrés).

5.2 Modèle de données

Comme on peut le voir ci-dessous, le modèle de données sera au format JSON qui est un format semi-structuré, ce qui signifie qu'il permet de combiner des données organisées et des champs plus flexibles, tout en restant lisible pour les humains et facilement exploitable.

Ce choix est adapté ici car il permet de regrouper dans un seul objet des informations hétérogènes : descriptives, catégorielles, fichiers multimédias et références vers des séries temporelles.

Mon modèle de données est structuré en quatre parties : **Description** (date de début, durée, fréquence en MHz, ZCR, modulation…), **labels** (genre du locuteur, identifiant, langues pour définir les éléments à cibler lors des attaques), **output** (liens vers fichiers audio et images spectrales) et **extracted physical metadata** (lien vers la table de la base de données contenant les courbes de l'énergie RMS, fréquence fondamentales et centroïde spectral associé à chaque signal audio collectés, ce sont des notions d'analyse de signal qui ne seront pas développé dans ce rapport).

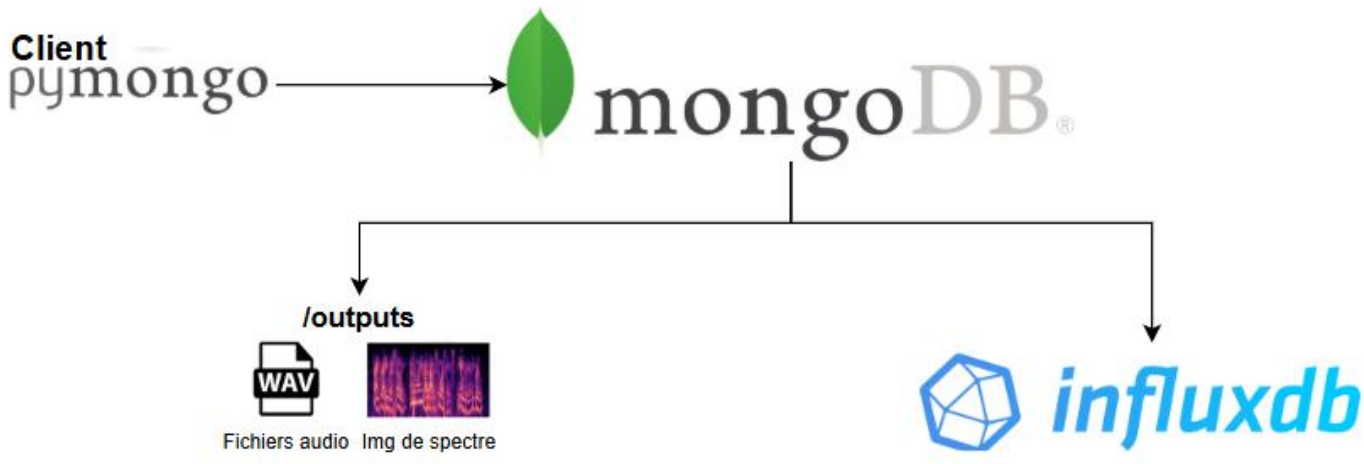
Dans ce modèle de données, les champs *output* sont défini à *null* car ils correspondent à des informations qui seront générées ultérieurement lors de la phase de transformation des données. Ces éléments constitueront les données d'entrée pour des tâches de classification en Deep Learning destiné à réaliser des tests d'attaques SCA dans un futur projet.

Ce modèle de données en JSON favorise une organisation claire des données en regroupant de manière structurée des informations hétérogènes, garantit leur intégrité grâce à un typage cohérent et à la présence explicite de valeurs nulles pour les champs pas encore disponibles. Cela offre une souplesse de manipulation grâce à sa compatibilité avec des langages de traitement, en particulier les langages de requête NoSQL et les échanges via HTTP, ce qui facilite son intégration dans des systèmes distribués et des bases orientées documents.


```json
{
  "description": {
    "recording_id": "rec_20250808_001",
    "datetime_start": "2025-07-08T14:23:05Z",
    "duration_seconds": 5.0,
    "center_frequency_mhz": 102.5,
    "zero_crossing_rate": 0.042,
    "modulation": "FM",
    "radio_namle": "Europe_1",
    "sample_rate_hz": 16000
  },
  "labels": {
    "speaker_gender": "male",
    "speaker_id": "spk_07",
    "languages": "fr"
  },
  "output": {
    "audio_file_url": "null",
    "spectrogram_image_url": "null"
  },
  "feature_physical_metadata": {
    "influxdb_table_url": "http://0.0.0.0:8181/api/v3/configure/table",
    "features": [
      "RMS_energy_timeseries",
      "fundamental_frequency_timeseries",
      "spectral_centroid_timeseries"
    ]
  }
}
```


6 Choix des solutions de stockage des données (C1.2.2)

6.1 Schéma et choix des technologies de stockage



J'ai choisi **MongoDB** comme base documentaire centrale pour stocker les métadonnées descriptives et catégorielles des signaux (fréquence, modulation, labels, liens vers fichiers ect.). Son langage de requête **MQL**, basé sur JSON, est intuitif, largement connu et adapté aux données semi-structurées, ce qui facilite l'intégration et l'évolution du schéma. Le format BSON et l'indexation native assurent des recherches rapides pour l'analyse et la comparaison d'un volume de document élevés, essentielles dans notre contexte.

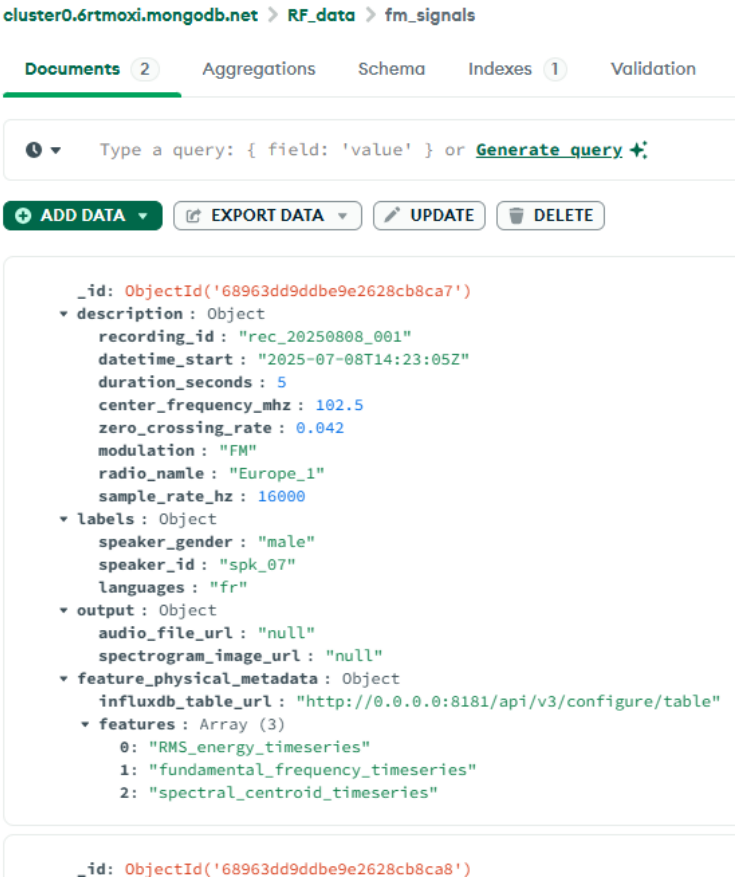
J'ai choisi de stocker sur disque les fichiers audio (**.WAV**) et spectrogrammes (.**PNG**). Pour l'audio, le format WAV non compressé garantit l'intégrité et la qualité du signal sans perte de données, indispensable pour conserver la totalité des informations exploitables. Chaque extrait audio contient plusieurs centaines de milliers de points, ce qui rend son stockage direct en base, même orientée séries temporelles, peu adapté en raison du volume et de la granularité, qui entraîneraient une surcharge due à l'indexation. Pour les images de spectre, le choix du format PNG permet de générer des représentations fréquentielles adaptées aux modèles de vision profonde, les réseaux de neurones convolutifs (deep learning) étant particulièrement efficaces pour la classification de ce type de données. La transformation d'un signal en image de spectre (appelé aussi spectrogramme) sera expliquée dans la suite de ce rapport.

J'ai choisi **InfluxDB** pour stocker les métadonnées correspondant aux caractéristiques physiques extraites des signaux (RMS, fréquence fondamentale, centroïde spectral) sous forme de séries temporelles. Son moteur est optimisé pour l'ingestion rapide et la requête sur de grands volumes de données chronologiques. Le langage **InfluxQL** (très similaire au SQL), plutôt connu dans l'IoT et l'analyse de mesures, facilite l'agrégation et la comparaison de séries issues de signaux RF. Contrairement aux fichiers audios complets, ces données sont synthétiques et réduites, ce qui les rend adaptées à un stockage en base orientée séries temporelles.

Le client **PyMongo** constitue le point d'entrée applicatif pour toutes les opérations CRUD, centralisant la connexion à MongoDB et l'accès aux collections de métadonnées.
Le design pattern retenu est de type **Polyglot Persistence** associé à un data lake léger mais scalable comme MongoDB qui gère les métadonnées, le stockage fichier conserve les WAV/PNG et InfluxDB héberge les caractéristiques temporelles de chaque fichier audio.

6.2 Extraits de la base documentaire centrale

J'utilise MongoDB Compass pour administrer la base et explorer les collections, ce qui me permet de parcourir visuellement les documents JSON, d'exécuter des requêtes MQL et des agrégations, de vérifier les index, ainsi que de contrôler les règles de validation de schéma. L'exemple affiché à côté illustre la structure type d'un enregistrement, avec l'horodatage de la capture, la fréquence en MHz, la fréquence d'échantillonnage, les chemins vers les fichiers WAV et PNG, les labels et les indicateurs de qualité comme le ZCR. Cette visualisation facilite le contrôle qualité et la traçabilité pendant l'ingestion, tout en confirmant que les liens vers les fichiers et les métadonnées sont cohérents.



6.3 Extraits de la base de métadonnées de séries temporelles (C1.2.2 & C1.3.2)

J'utilise InfluxDB 3 Core comme base orientée séries temporelles pour les mesures dérivées de mes signaux audio. InfluxDB est adapté ici car il offre une ingestion rapide, des requêtes efficaces par fenêtres de temps, et des règles de rétention, ce qui convient parfaitement à des indicateurs calculés à haute fréquence. C'est à ce jour la base de données la plus performante en termes d'écriture/lecture/compression pour des données horodatés. Concrètement, je stocke les caractéristiques compactes et temporellement associé à chaque fichier audio (RMS, fréquence fondamentale et centroïde spectral).

```python
HOST  = "http://localhost:8181"
DB    = "rf"
TOKEN = Token
# Tags
RECORD_ID = "record_20250808_001"
MONGO_ID  = "66abcdeffedcba1234567890"
client = InfluxDBClient3(host=HOST, database=DB, token=TOKEN)
def write_series(measurement: str, t_sec: np.ndarray, y: np.ndarray):
    points = []
    base_ns = time.time_ns()
    for t, v in zip(t_sec, y):
        if np.isnan(v):
            continue
        ts_ns = base_ns + int(t * 1e9)  #  nanosecondes
        p = (
            Point(measurement)
            .tag("record", RECORD_ID)
            .tag("mongoid", MONGO_ID)
            .field("value", float(v))
            .time(ts_ns, write_precision="ns")
        )
        points.append(p)
```

Ci-contre j'utilise le client Python officiel InfluxDB (influxdb3-python) pour sérialiser les mesures et les envoyer à l'API HTTP v3 d'InfluxDB, avec authentification par token.

Voici ci-dessous un exemple en Line Protocol de plusieurs points dans influxdb3. Ici un point de mesure dans la série RMS avec deux tags d'identification (record pour le numéro de capture et mongoId pour le lien vers le document MongoDB). Le champ value contient la valeur RMS 0.12 pour l'intervalle analysé, et le timestamp 1755255726000000000 est en nanosecondes. Cela permet à InfluxDB de classer, filtrer et agréger précisément cette mesure dans le temps et par métadonnées.

```
rms,record=record_20250808_001,mongoid=66abcdeffedcba1234567890 value=0.12   1755255726000000000
rms,record=record_20250808_001,mongoid=66abcdeffedcba1234567890 value=0.18   1755255726010000000
rms,record=record_20250808_001,mongoid=66abcdeffedcba1234567890 value=0.24   1755255726020000000
```

Pour terminer voici une capture ci-dessous de Data Explorer, l'interface Web d'InfluxDB 3 qui permet de se connecter au serveur, parcourir les bases et exécuter des requêtes InfluxQL. Je peux désormais consulter les métadonnées de série temporelle associées à mon audio précédemment enregistré par exemple ici centroïde spectral. Je peux également filtrer par tags (record, mongoid) et par période, visualiser les points, puis lancer directement des requêtes SQL sur la base Influx pour analyser et agréger ces mesures. Cela facilite la traçabilité et le contrôle qualité de chaque capture.



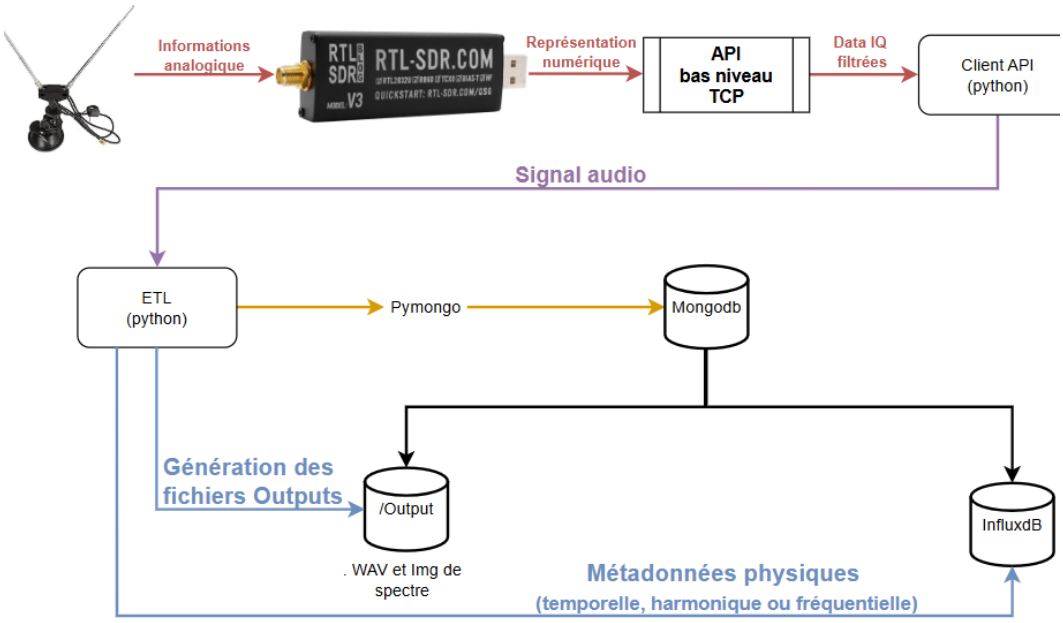
7 Traitement de données (C1.3.1)

7.1 Langage et pipeline de traitement de données.

J'ai choisi d'utiliser Python comme langage principal pour l'ensemble des traitements de données et de signaux dans le projet. Il me permet à la fois de réaliser les requêtes vers la base MongoDB pour récupérer les métadonnées, de traiter les signaux RF, d'extraire des caractéristiques audio (RMS, fréquence fondamentale, centroïde spectral) et de transformer les données en formats exploitables (WAV, spectrogrammes PNG).

Ce choix est particulièrement adapté dans mon contexte car Python dispose d'un écosystème scientifique très complet qui couvre l'ensemble des besoins, calcul numérique performant, traitement du signal, analyse audio et visualisation. De plus, sa lisibilité et sa rapidité de développement facilitent la mise en place d'un pipeline complexe tout en restant flexible pour intégrer de nouvelles étapes ou méthodes d'analyse au fur et à mesure des expérimentations.

Ci-contre, mon schéma synthétique de traitement des données illustrant les étapes clés, depuis la collecte du signal RF jusqu'à la production des fichiers et métadonnées exploitables. J'utilise python dès le moment où je reçois les données numériques du SDR, jusqu'à l'écriture des données dans la base documentaire (cheminement en Jaune), en passant par le traitement de signal audio (cheminement en violet) ainsi que pour toutes les étapes de transformation de données (cheminement bleu).



7.2 Choix des librairies utilisés

Comparatif des technologies de traitement de données sélectionnées

Technologie	Avantages	Inconvénients	Utilisation
Python	Polyvalent, vaste écosystème scientifique, syntaxe lisible, communauté très active	Moins rapide que C/C++ pour traitement en temps réel	Couvre toutes les étapes, requêtes BDD, traitement RF, audio, visualisation
NumPy	Calcul vectorisé rapide (implémenté en C), gestion efficace des grands tableaux de données	Nécessite d'adopter une logique vectorisée	Manipulation performante des flux I/Q RF
SciPy.signal	Large choix de filtres et outils, fiable pour traitement audio/RF	Peut devenir lent sur très gros volumes	Filtrage, démodulation et transformations fréquentielles
Librosa	Extraction facile de features audio (RMS, MFCC, spectrogrammes)	Moins adapté au traitement en flux temps réel	Extraction de caractéristiques exploitables en ML/Deep Learning
PyMongo	Interface Python native pour MongoDB, requêtes MQL simples à exécuter, gestion facile des documents JSON	Nécessite de connaître la structure des collections pour optimiser les requêtes	Récupération rapide et ciblée des métadonnées de signaux
InfluxDB Client (Python)	Compatible InfluxDB v2 et v3, écritures/lectures optimisées, gestion native des séries temporelles	Configuration initiale et gestion d'authentification	Envoi structuré et rapide des mesures RF vers InfluxDB
Socket (Python)	Simple à mettre en œuvre pour recevoir un flux TCP	Ne gère pas nativement des protocoles complexes	Réception en continu des données I/Q depuis le SDR

8 Présentation des données transformées (C1.3.2)

Les transformations de données ici concernent la génération des fichiers Outputs (les transformations qui concernent l'extraction des caractéristiques physique du signal insérées dans Influxdb ne seront pas détaillées dans ce rapport). Nous avons donc ici la présentation des transformations du signal audio en fichier .WAV normalisés et en versions chiffrées (algorithme Simple Scrambling), ainsi que la création d'images spectrogrammes associés.

8.1 Transformation signal audio au format WAV

Une fois le signal audio traité (filtrage, démodulation, rééchantillonnage, normalisation), il est exporté au format WAV via *soundfile.write*. La fréquence d'échantillonnage est de 16 000 Hz, ce qui est largement suffisant ici pour de l'audio (la plupart des micros ne montent pas au-dessus de 12Khz), la qualité ici est 16 bits ce qui correspond à 65 536 valeurs possibles. On utilise donc ici le format PCM (Pulse Code Modulation) 16 bits, PCM une méthode d'encodage audio non compressé très connu. Ce choix garantit un stockage sans perte d'information utile, une interopérabilité maximale avec les outils et autre format audio et des opérations de lecture/écriture rapides sur disque. Le fichier est nommé de manière à intégrer des informations clés (horodatage, fréquence centrale) et son chemin est enregistré dans la base documentaire, ce qui assure la traçabilité et permet la jointure avec d'autres artefacts du même enregistrement (spectrogrammes, mesures physiques). Par exemple il sera facile de rechercher tous les enregistrements en français captés à 102 MHz dans MongoDB et d'avoir l'ID correspondant pour retrouver leurs .WAV, leurs spectrogrammes et leurs courbes de RMS dans InfluxDB.

```python
import soundfile as sf

# 'resampled' contient déjà le signal audio prêt à être sauvegardé
AUDIO_RATE = 16000  # fréquence d'échantillonnage en Hz

# Nom de fichier : date_heure_fréquence.wav
output_filename = "output/rec_20250808_001.wav"

# Sauvegarde au format WAV, PCM 16 bits
sf.write(output_filename, resampled, AUDIO_RATE, subtype='PCM_16')

print(f"Fichier audio enregistré : {output_filename}")
```

8.2 Transformation signal audio au format WAV chiffré

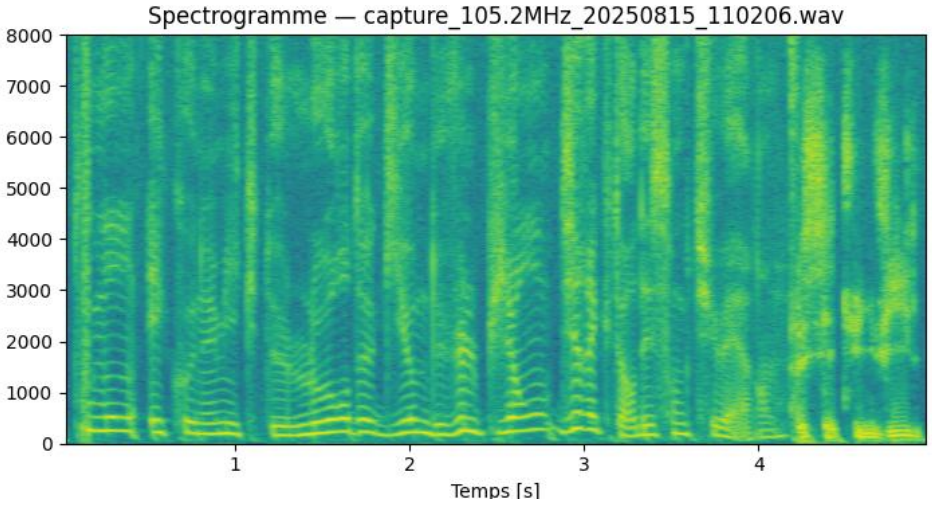
Ce traitement applique un algorithme de brouillage fréquentiel (Simple Scrambling) au fichier WAV initial afin de créer une version chiffrée pédagogique. La méthode utilisée consiste à convertir le signal audio en signal analytique (via transformée de Hilbert), puis à effectuer une translation fréquentielle définie par une clé fc connue. Cette transformation est réversible si la clé est connue, mais rend le contenu audio inintelligible à l'oreille. Le fichier résultant est exporté en PCM 16 bits, avec un nom distinctif (suffixe _scr), ce qui permet de le distinguer de la version originale.

```python
import numpy as np
import soundfile as sf
from scipy.signal import hilbert
AUDIO_RATE=16000
FC=4000.0
t=np.arange(len(resampled))/AUDIO_RATE
analytic=hilbert(resampled)
scrambled=np.real(analytic*np.exp(2j*np.pi*FC*t))
scrambled/=np.max(np.abs(scrambled)+1e-12)
sf.write("rec_20250808_001_src.wav",scrambled,AUDIO_RATE,subtype='PCM_16')
```

8.3 Spectrogramme et jointure temps-fréquence.

Pour générer un spectrogramme, on charge l'audio (WAV), on le passe en mono puis on découpe le signal en petits segments qui se recouvrent et j'analyse pour chacun "quelles fréquences sont présentes". En assemblant ces résultats, on obtient une carte temps–fréquence lisible avec l'axe horizontal qui représente le temps (en secondes), l'axe vertical la fréquence (en hertz) et la couleur représente l'énergie du signal (en dB). Si le haut de l'axe s'arrête à 8 kHz, c'est simplement parce que l'audio est échantillonné à 16 kHz, on ne peut pas voire au-delà de la moitié de la fréquence d'échantillonnage (limite de Nyquist).

Ces images de spectre pourront ensuite être exportées et pourront servir d'entrées pour entraîner et tester des modèles de deep learning dans le cadre d'attaques par canaux auxiliaires (SCA), afin d'évaluer la capacité des réseaux (CNN, etc.) à reconnaître des signatures ou des motifs caractéristiques liés aux signaux chiffrés.



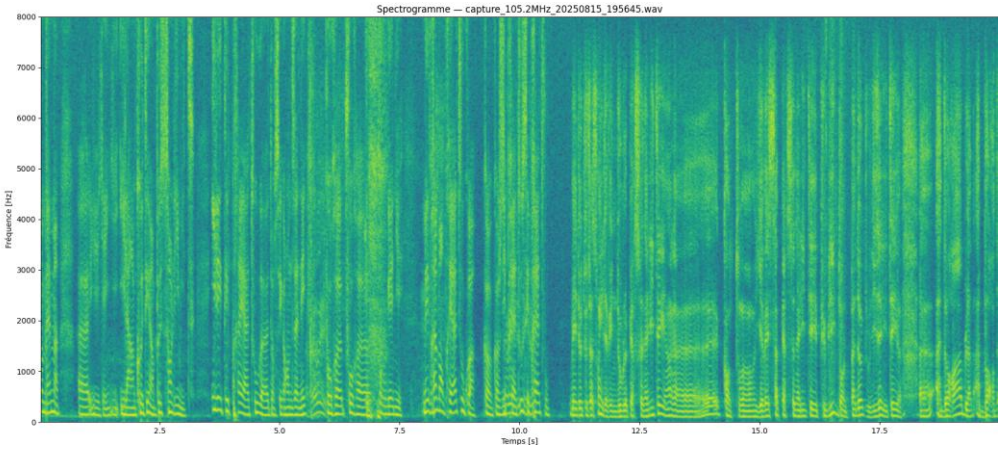

```python
# Normalisation simple (évite les échelles bizarres)
m = np.max(np.abs(audio))
if m > 0:
    audio = audio / m

# --- Spectrogramme (STFT) ---
# fenêtre Hann ~64 ms (nperseg=1024 à 16 kHz)
nperseg = 1024
noverlap = 768
f, t, S = signal.spectrogram(
    audio,
    sr,
    window="hann",
    nperseg=nperseg,
    noverlap=noverlap,
    nfft=nperseg,
    scaling="density",
    mode="magnitude",
)

# Passage en dB
S_db = 20 * np.log10(S + 1e-12)

# --- Affichage + sauvegarde ---
plt.figure(figsize=(10, 4))
plt.pcolormesh(t, f, S_db, shading="gouraud")
plt.title(f"Spectrogramme — {Path(WAV_PATH).name}")
plt.xlabel("Temps [s]")
plt.ylabel("Fréquence [Hz]")
plt.tight_layout()
```

Pour certaines attaques SCA, comme détecter la langue sur des signaux chiffrés, il faut des séquences plus longues que 5 s, par exemple 20 s ou plus. Pour y parvenir, je joins temporellement plusieurs extraits de 5 s : je charge les deux audios, les harmonise en mono 16 kHz, puis je concatène les signaux. Une fois le signal allongé obtenu, je calcule un seul spectrogramme continu avec les mêmes paramètres que d'habitude, ce qui conserve un axe temps cohérent pour l'analyse et l'entraînement de modèles de deep learning. J'enregistre en parallèle les métadonnées. Un point très Important ici, je ne "recolle" pas des images de spectrogrammes entre elles, car pour les algorithmes de deep learning il faut respecter un format d'image fixe (dimensions, résolution), donc on concatène d'abord l'audio, puis on génère une image unique recadrée aux dimensions attendues par le modèle.



9 Talend pour l'ingestion et la transformation de données (C1.3.3)

Talend est un atelier d'intégration qui assemble des flux ETL à partir de composants prêts à l'emploi (fichiers, bases, REST, etc.). Dans notre contexte RF / attaques SCA, Talend sert d'"orchestrateur" entre trois mondes : les métadonnées (MongoDB), les métadonnées de séries temporelles (InfluxDB 3 Core) et la transformation calculatoire (scripts Python pour spectrogrammes et features).

Avantages dans notre cas d'usage

- **Connecteurs out-of-the-box** : MongoDB (tMongoDBOutput), REST générique pour InfluxDB 3 (tRESTClient), fichiers CSV/JSON/Excel (planning, features).
- **Orchestration low-code** : enchaîner collecte → transformation → persistance, gérer les erreurs (on-component error), journaliser et planifier.
- **Gouvernance / Intégrité** : upsert par clé fonctionnelle (record), validation côté base (JSON Schema Mongo), séparation claire blobs / métadonnées / timeseries.
- **Extensibilité** : la transformation lourde reste en Python (FFT/STFT, features) et Talend ne fait que la piloter et router les résultats.

9.1 Job_Mongo_Ingest métadonnées dans MongoDB

Chaîne tFileInputJSON/Delimited → tMap → tMongoDBOutput :

- Le tMap normalise les champs (ex. paths.wav, quality.zcr, quality.keep), typage cohérent.
- tMongoDBOutput est configuré en Upsert sur la clé record (Action on data : Upsert, Upsert key : record). Cela garantit l'idempotence (rejouer le job).
- Côté base, on applique un JSON Schema (dans Mongo) pour verrouiller l'intégrité (champs requis, types, bornes).

Exemple de document inséré :

```
{
  "record": "record_20250808_001",
  "ts": ISODate("2025-08-15T11:02:06Z"),
  "freq_mhz": 105.2,
  "sr": 16000,
  "paths": {"wav": "capture_105.2MHz_20250815_110206.wav", "png": null},
  "quality": {"zcr": 0.014, "keep": true},
  "mongoid": "66abcdeffedcba1234567890"
}
```

9.2 Job_Influx_Write métadonnées de séries temporelles vers InfluxDB 3

Chaîne : tFileInputDelimited → tJavaRow/tMap → tRESTClient

- On part d'un CSV colonne-based (measurement,record,mongoid,value,ts_ns).
- Dans tJavaRow, on construit la ligne Line Protocol dans une colonne lp qui sera envoyée telle quelle au POST.

```
// In: measurement, record, mongoid, value, ts_ns | Out: lp (String)
String meas    = input_row.measurement;      // "rms" / "f0" / "centroid"
String record  = input_row.record;           // "record_20250808_001"
String mongoid = input_row.mongoid;          // "66abcdeffedcba1234567890"
Double value   = input_row.value;            // ex: 0.12
Long ts_ns     = input_row.ts_ns;            // epoch ns

if (value == null || value.isNaN()) {
  output_row.lp = null; // filtrée ensuite par tFilterRow
} else {
  output_row.lp = meas + ",record=" + record + ",mongoid=" + mongoid
                + " value=" + String.valueOf(value) + " " + String.valueOf(ts_ns);
}
```

tRESTClient :

- Method : POST
- URL : ${context.influx_url}/api/v3/write_lp?db=${context.influx_db}&precision=ns
- Headers : Authorization: Bearer ${context.influx_token}, Content-Type: text/plain; charset=utf-8
- Body : la colonne lp

Exemple de ligne produite :
rms,record=record_20250808_001,mongoid=66abcdeffedcba1234567890 value=0.12 1755255726000000000

9.3 Job_Transform spectrogramme & features via Python

Chaîne : tFileList (capture_*.wav) → tSystem/tPython (script) → tExtractJSONFields → tMongoDBOutput (+ tRESTClient optionnel)

- tFileList parcourt les WAV. tSystem ou tPython appelle le script Python de transformation, qui produit un PNG et imprime un JSON minimal sur stdout.
- tExtractJSONFields parse le JSON du script pour récupérer wav, png, sr, etc.
- tMongoDBOutput met à jour le document (upsert par record) pour remplir paths.png, ajouter des mesures qualité, etc.
- Si le script calcule aussi RMS/F0/centroïde, on peut dériver les CSV/flux nécessaires et réutiliser le job 2 (ou appeler un tRESTClient rapide).

Extrait Python (appelé par Talend) :

```
wav_path = Path(args.inp)
audio, sr = sf.read(wav_path, always_2d=False)
if getattr(audio, "ndim", 1) > 1: audio = audio[:, 0]
m = np.max(np.abs(audio));  audio = audio/m if m>0 else audio

f, t, S = signal.spectrogram(audio, sr, window="hann", nperseg=1024, noverlap=768, nfft=1024, mode="magnitude")
S_db = 20*np.log10(S + 1e-12)
plt.pcolormesh(t, f, S_db, shading="gouraud"); plt.tight_layout(); plt.savefig(args.out_png, dpi=150); plt.close()

print(json.dumps({"wav": str(wav_path), "png": str(Path(args.out_png).resolve()), "sr": int(sr)}))
```

En contexte polyglotte comme le nôtre avec métadonnées en MongoDB, métadonnées de séries temporelles avec InfluxDB et fichiers, Talend est un très bon choix. Ses connecteurs natifs et son orchestration low code évitent le glue code tout en garantissant idempotence, intégrité et traçabilité des flux. Il s'adapte bien à beaucoup de transformations comme spectrogrammes, features et contrôle qualité en chaînant simplement des composants avec des scripts Python, des contexts pour changer d'environnement et une gestion d'erreurs et de relances intégrée. Côté exploitation il permet de faire de l'ingestion quasi-temps réel et de livrer rapidement des données prêtes à l'analyse pour les ingénieurs métiers avec un monitoring clair et une montée en charge progressive. Talend industrialise notre pipeline multi bases et orienté calcul tout en restant lisible robuste et évolutif.

10 Politique de sécurité (C1.4.1)

Notre projet traite des signaux RF, d'audio et de spectrogrammes, avec métadonnées dans MongoDB et séries temporelles dans InfluxDB. Les risques majeurs portent sur la confidentialité des contenus audio, l'intégrité des données utilisées pour l'IA, et la disponibilité des pipelines.

Il faudra classer l'audio et les spectrogrammes en niveau élevé, les features agrégées comme RMS, F0 et centroïde en niveau moyen, et les paramètres techniques ainsi que les journaux en niveau standard. On devra appliquer systématiquement la minimisation et la pseudonymisation à l'aide d'identifiants de capture comme record et mongoid, il faudra penser à limiter la durée de conservation aux besoins scientifiques, et tracer chaque transformation. Exemple concret, lorsqu'une séquence contient de la parole, le lien entre l'enregistrement et une identité réelle n'est jamais stocké et la durée de conservation est réduite.

Les droits d'accès sont définis par rôles selon le principe du moindre privilège. Les opérateurs de collecte écrivent en bronze mais ne modifient ni schémas ni index, les data engineers gèrent les jobs Talend, la qualité des données, les schémas et la rétention, les data scientists lisent les jeux silver et gold et disposent d'un token InfluxDB en lecture seule pour analyser et visualiser.

Côté bases, MongoDB fonctionne avec des comptes applicatifs dédiés, RBAC par collection, validation JSON Schema, index d'unicité sur record et writeConcern majority, InfluxDB 3 utilise des tokens à privilège minimal distincts pour écriture et lecture, le stockage fichiers applique des permissions séparées par zone bronze, silver, gold. Talend orchestre les flux avec contexts dev, test, prod, pas de secrets en clair et journalisation de chaque exécution. Exemple concret, une connexion anormale ou un pic d'erreurs d'écriture déclenche une alerte, la révocation du token et l'ouverture d'un ticket d'incident.

On chiffre les échanges avec MongoDB et InfluxDB via TLS, et les fichiers sont chiffrés au repos. On segmente le réseau, on isole les services en conteneurs, et on vérifie l'intégrité des WAV/PNG avec des empreintes SHA-256.

Des sauvegardes régulières avec tests de restauration sont en place et la rétention est adaptée à chaque zone, bronze courte, silver moyenne, gold plus longue et plus anonymisée. La conformité RGPD repose sur une finalité de recherche documentée, une DPIA si traitement de voix, des mécanismes pour répondre aux droits des personnes et des durées de conservation maîtrisées. Cas concrets, perte d'un ordinateur contenant des extraits, les fichiers sont chiffrés et les tokens révoqués immédiatement, départ d'un collaborateur, rotation des secrets et désactivation de ses comptes, erreur d'ingestion sur un lot, on identifie les records via les journaux, on purge et on rejoue le flux.

Checklist opérationnelle concrète

• Rétention, audio bronze 30 jours, spectrogrammes PNG 90 jours, features InfluxDB 12 à 24 mois, purge automatique au-delà

• Tokens InfluxDB par rôle, rotation trimestrielle, clés MongoDB et mots de passe stockés dans un coffre de secrets, journalisation des accès et des rejets

• Politique de nommage et d'isolement des tables Influxdb et répertoires, un bucket par environnement et par zone, droits lecture seule pour les data scientists sur gold

• Qualité de données à l'ingestion, validation ZCR, RMS min et pourcentage de clipping max, rejet automatique en KO avec log détaillé et identifiant de lot

• Intégrité, calcul du hash avant écriture, stockage de l'empreinte dans MongoDB, vérification du hash avant tout entraînement de modèle

• Continuité de service, sauvegarde quotidienne MongoDB et snapshots périodiques InfluxDB, test de restauration mensuel, supervision de Talend avec alerting

• Conformité, procédure simple pour les demandes de suppression liée à un speaker_id, documentation des finalités et mise à jour annuelle de l'analyse d'impact.

Avec ces dispositions les enjeux de sécurité sont identifiés, les moyens de protection sont concrets et appliqués, et l'intégrité, la disponibilité et la bonne utilisation des données sont garanties au regard des risques et des obligations légales.

11 Conclusion

En résumé, ce projet met en place une chaîne complète d'ingestion et de préparation de données RF et audio, de la capture en quasi-temps réel jusqu'au stockage structuré et sécurisé. Les signaux sont démodulés, contrôlés en qualité, transformés en représentations utiles comme les spectrogrammes et les features, puis organisés dans des stockages adaptés avec MongoDB pour les métadonnées et InfluxDB pour les séries temporelles, le tout orchestré par Talend.

Ces données sont complexes à traiter et à analyser, elles demandent des choix techniques rigoureux dès la phase de data engineering pour garantir intégrité, traçabilité et performance. Ce socle rend possible une exploration de données solide dans un projet dédié à l'analyse exploratoire et la visualisation, puis ouvre la voie à un autre projet focalisé sur les attaques SCA par apprentissage profond, où l'on pourra entraîner et évaluer des modèles sur des jeux de données fiables et bien gouvernés.