PROJET STATION TV

Dossier de Conception Détaillé

SUIVI DES MODIFICATIONS DU DOCUMENT			
Version	Date	Validation	Commentaire
1.0	19/11/2025	Frédéric Chauvin	Rédaction initiale

REDACTEUR : Dorian BRISSON	VALIDATEUR : Frédéric CHAUVIN
CLIENT : Polytech Tours	ENCADRANT : Mathieu DELALANDRE

Table des matières

PROJET STATION TV 1

OBJET DU DOCUMENT 4
Objectif du document 4
Position dans la démarche de développement 4
Périmètre de la conception détaillée 5
ARCHITECTURE LOGICIELLE 5
Structure globale du projet 5
Diagramme de classes 6
Description des packages 7
Schéma logique de l'architecture 9
DESCRIPTION DES MODULES 9
Module Preprocessing 9
Module Core Transcription 10
Module QoS 11
Module Export 11
ALGORITHMES INTERNES 11
Vue générale 11
Algorithme d'équilibrage de charge 12
Gestion de l'affinité CPU 13
Gestion mémoire adaptative 13
Transcription d'un fichier audio 14
Monitoring QoS en temps réel 15
Calcul du throughput 15
Calcul du Word Error Rate (WER) 16
Contexte 16
Export incrémental sécurisé 16
Stratégie de tolérance aux erreurs 17
GESTION DES DONNEES ET FORMATS 17
Vue générale 17
Formats d'entrée 18
Données intermédiaires 18
Formats de sortie 19
Modèles de données internes 21
Données de support 21
Sécurité des données 22
Conformité et traçabilité 23
TESTS UNITAIRES 23
Objectifs des tests unitaires 23
Environnement de test 24
Stratégie de test 24
Modules testés 25
Critères de réussite 27
Gestion des erreurs en test 27
Processus d'exécution 27
Reporting 28
Intégration avec pipeline Agile 28
GESTION DES ERREURS ET SECURITE 28
Objectifs généraux 28
Sources de risques identifiées 29
Stratégie générale de tolérance aux erreurs 30
Isolation mémoire et sécurité RAM 31

Sécurité des données *31*
Gestion des erreurs système *32*
Exemple *32*
Logging et auditabilité *32*
Sécurité opérationnelle *33*
Exemple de protection : *33*
Résilience du pipeline aux interruptions *33*
Robustesse globale *34*
DEPLOIEMENT ET AUTOMATISATION 34
Objectifs du déploiement *34*
Prérequis matériels *34*
Prérequis logiciels *35*
Installation *35*
Configuration *36*
Automatisation de l'exécution *36*
Automatisation des tests *37*
Automatisation du monitoring *37*
Gestion des erreurs automatisée *37*
Modes d'exécution *38*
Optimisation du déploiement *38*
Sécurité opérationnelle *39*
Reproductibilité *39*
Étapes complètes de mise en production *39*

OBJET DU DOCUMENT

Objectif du document

Le présent document constitue le Dossier de Conception Détaillé (DCD) du projet Station TV – Système de transcription audio haute performance.

Son objectif est de fournir une description technique approfondie des éléments suivants :

- La **structure logicielle** du projet,
- Les **modules de traitement** et leurs responsabilités,
- Les **interfaces internes** et flux de données,
- Les **algorithmes d'orchestration et de parallélisation**,
- Les **formats structurés** utilisés (JSON, CSV, SRT),
- Les **mécanismes de monitoring et QoS**,
- Les **tests unitaires et d'intégration**.

Ce document sert de référence :

- Pour le développement, la maintenance et la réutilisation du système,
- Pour l'audit technique du projet,
- Pour l'extension future des fonctionnalités.

Position dans la démarche de développement

Le DCD intervient après la rédaction du DCG et la validation des documents de cadrage. Il a pour rôle de décrire précisément la mise en œuvre des composants logiciels, en cohérence avec les objectifs fonctionnels et non fonctionnels.

Il représente le niveau de spécification nécessaire à l'implémentation, et constitue un support pour :

- Les sprints de développement,
- La revue de code,
- La validation technique,
- Le transfert de connaissances.

Périmètre de la conception détaillée

Ce document couvre :

- Le code Python du pipeline Station TV,
- Les modules Preprocessing, Core, Export, QoS, Utilities,
- Les structures de données,
- Les algorithmes internes,
- Les interfaces entre modules.

Sont exclus :

- Les analyses fonctionnelles générales (décrites dans le DCG),
- Les spécifications matérielles,
- Les résultats de performance et leur analyse détaillée.

ARCHITECTURE LOGICIELLE

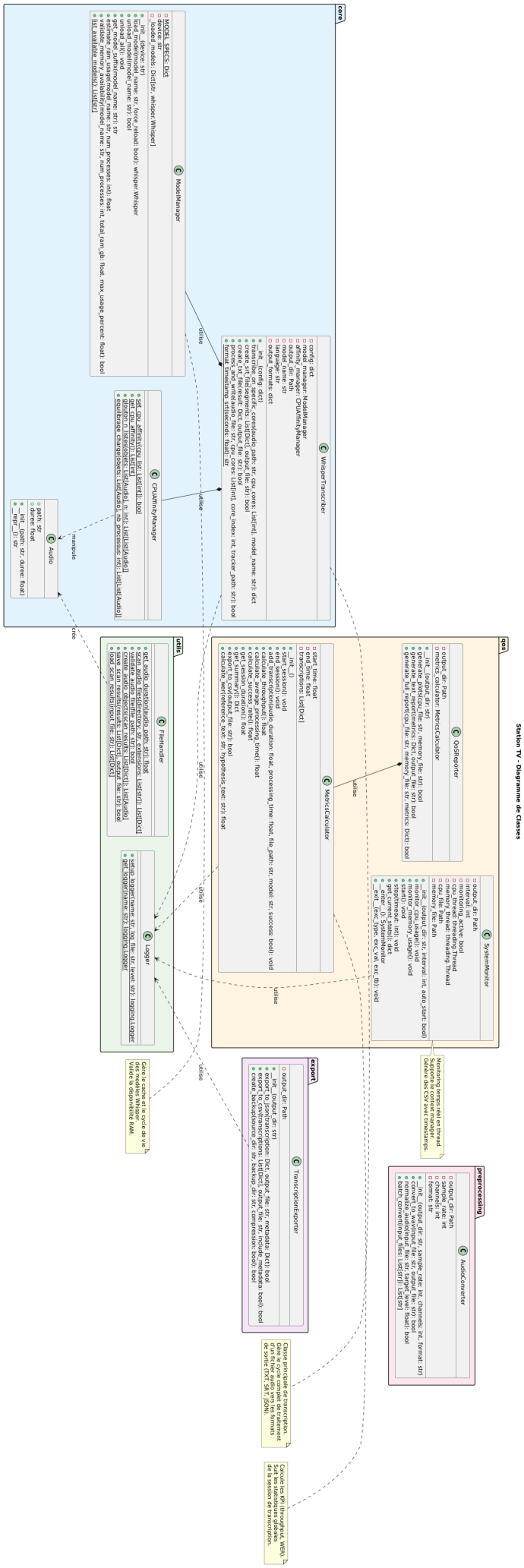
Structure globale du projet

Le code source est organisé selon une structure modulaire :

```
stationtv/
 ├── config/
 ├── core/
 ├── qos/
 ├── preprocessing/
 ├── export/
 ├── utils/
 ├── scripts/
 ├── tests/
 └── requirements.txt
```

Chaque répertoire correspond à un module fonctionnel indépendant.

Diagramme de classes



Description des packages

Package preprocessing/

Contient les fonctions de préparation audio :

- Conversion MP3 → WAV
- Normalisation du signal
- Segmentation optionnelle
- Extraction de métadonnées

Fichiers principaux :

- audio_converter.py

Package core/

Implémente la transcription et l'orchestration multiprocess.

Responsabilités :

- Gestion des modèles Whisper,
- Parallélisation CPU,
- Équilibrage de charge,
- Traitement des segments,
- Émission des métadonnées.

Fichiers :

- transcription.py
- models.py
- affinity.py

Package qos/

Responsable du monitoring et de l'analyse de performance.

Fonctionnalités :

- Mesure CPU/RAM périodique,
- Calcul throughput,
- Calcul WER,
- Export CSV + graphiques PNG.

Fichiers :

- monitor.py

- metrics.py
- reporter.py

Package export/

Gère la génération des sorties finales :

- TXT / JSON / CSV / SRT
- sauvegardes automatiques
- intégration dans Station TV

Fichier :

- exporter.py

Package utils/

Contient les utilitaires transverses :

- logging,
- gestion fichiers et dossiers,
- chargement de configuration,
- parsing CSV.

Fichiers :

- logger.py
- file_handler.py

Package scripts/

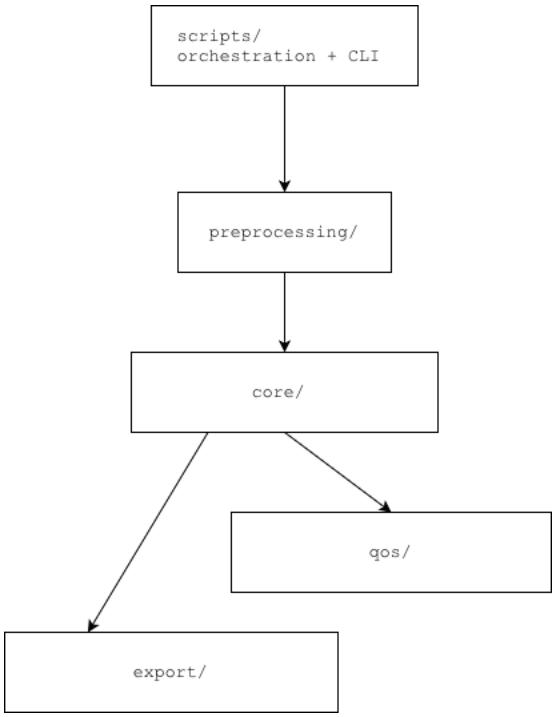
Scripts d'orchestration :

- exécution pipeline complet,
- tests unitaires,
- batch processing.

Fichiers :

- RunPipeline.py
- RunTests.py

Schéma logique de l'architecture



DESCRIPTION DES MODULES

Module Preprocessing

Responsabilités

- standardiser les fichiers audio
- réduire la complexité acoustique
- garantir compatibilité Whisper

Entrées / Sorties

Entrée	Sortie
MP3	WAV normalisé

Fonctions principales

def convert_mp3_to_wav(path_in, path_out):

def normalize_audio(wav_file):

def segment_audio(wav_file, duration=15):

Exceptions

- invalide audio format
- erreur FFmpeg
- durée non calculable

Module Core Transcription

Responsabilités

- Gérer les modèles
- Orchestrer les processus
- Générer segments + timestamps
- Éviter la saturation RAM

Fonctions majeures

def load_model(name):

def transcribe_file(path):

def run_batch(pool, files):

Algorithme multi-process (simplifié)

1: déterminer N processus
2: mesurer RAM par modèle
3: répartir fichiers par durée
4: lancer pool
5: surveiller erreurs

Module QoS

Métriques

- CPU usage
- RAM usage
- throughput
- vitesse moyenne
- WER

Exports

- CSV brut
- PNG graphiques
- rapport texte

Module Export

Formats générés

Format	Usage
TXT	lecture
JSON	structuration
CSV	analyse
SRT	sous-titres

Algorithmes internes

Vue générale

Les algorithmes internes du système Station TV répondent à quatre objectifs principaux :

1. **Maximiser le débit de transcription** sur CPU multicœur.
2. **Limiter la saturation mémoire** lors du chargement de modèles lourds.
3. **Assurer une stabilité forte** lors d'exécutions longues.
4. **Produire des données exploitables** avec traçabilité complète.

Pour répondre à ces contraintes, le système combine :

- du **multi-process CPU isolé**,
- une **surveillance périodique des ressources**,
- des **exports incrémentaux sécurisés**.

Algorithme d'équilibrage de charge

Objectif

Répartir un ensemble de fichiers audio entre N processus, en équilibrant les durées totales de traitement.

Problème :

- Durée inconnue a priori, mais approximable par longueur audio.
- Modèles Whisper non-streaming → mémoire proportionnelle aux threads.

Approche adoptée

Par tri décroissant :

1. Trier les fichiers par durée décroissante.
2. Créer N listes (une par processus).
3. À chaque itération :
    - affecter le fichier le plus long au processus le moins chargé.

Pseudo-code

```
files = sort_by_duration_desc(files)
processors = [[] for _ in range(N)]
load = [0]*N

for f in files:
   idx = argmin(load)
   processors[idx].append(f)
   load[idx] += duration(f)
```

Résultat

- Variance du temps de traitement minimisée
- Charge homogène

Gestion de l'affinité CPU

Objectif

Allouer des cœurs dédiés à chaque processus pour :

- limiter les conflits,
- éviter le thrashing cache,
- assurer une stabilité thermique et performance.

Approche

À chaque processus P :

1. Déterminer un sous-ensemble de cœurs disponibles.
2. Définir les masques d'affinité.
3. Appliquer via API OS.

Pseudo-code

```
cores = detect_logical_cores()

for p in processes:
  bind(p, cores[p.index])
  start(p)
```

Effet

- Isolation CPU par processus
- Réduction significative des fluctuations

Gestion mémoire adaptative

Contrainte

Le modèle Whisper Medium nécessite **>128Go RAM par instance**.
Whisper Large peut dépasser **>200 Go**.

Stratégie

1. Estimer RAM disponible.
2. Calculer RAM nécessaire par modèle.
3. Déterminer nombre maximal de processus.
4. Ajuster automatiquement N.

Pseudo-code

```
ram_total = detect_ram()
ram_needed = estimate_model_ram(model)
max_proc = floor(ram_total / ram_needed)

N = min(user_requested, max_proc)
```

Effet

- Crash impossibles par saturation,
- Auto-adaptation à l'environnement,
- Optimisation du débit sans risque.

Transcription d'un fichier audio

Objectif

Convertir un fichier WAV en segments textuels avec :

- timestamps,
- scores de confiance,
- format structuré.

Pseudo-code

```
model = load_model()
audio = load_audio(path)

segments = model.transcribe(audio,
        timestamps=True,
        word_timestamps=True)

for seg in segments:
   save(seg)
```

Structure segment

```
{
  "start": 12.96,
  "end": 15.78,
  "text": "Bonjour à tous",
  "confidence": 0.92
}
```

Monitoring QoS en temps réel

Objectif

Mesurer les métriques système **toutes les 30 secondes**.

Métriques enregistrées

- CPU %
- RAM %
- RAM absolue
- débit (x temps réel)
- vitesse moyenne
- WER

Pseudo-code

```
while running:
   cpu = psutil.cpu_percent()
   ram = psutil.virtual_memory().percent
   save_metrics(cpu, ram)
   sleep(30)
```

Effet

- Série temporelle cohérente
- Diagnostic a posteriori
- Alertes pilotées par données

Calcul du throughput

Définition

throughput = (durée audio traitée) / (durée réelle)

Objectif

- throughput ≥ 1.4× sur modèle Medium

Exécution

```
start = now()
audio_processed = 0

for f in files:
   audio_processed += duration(f)

elapsed = now() - start
throughput = audio_processed / elapsed
```

Calcul du Word Error Rate (WER)

Contexte

WER = indicateur de qualité de transcription.
Mesuré sur un **échantillon**.

Formule

WER = (S + D + I) / N

Avec :

- S = substitutions
- D = suppressions
- I = insertions
- N = nombre total de mots

Pseudo-code

```
def wer(ref, hyp):
   S, D, I = align(ref, hyp)
   return (S + D + I) / len(ref)
```

Export incrémental sécurisé

Objectif

Éviter perte de données en cas de crash.

Mécanisme

- sauvegarde par fichier traité,
- flush systématique,
- backups périodiques.

Pseudo-code

```
for file in processed:
   json_save(file)
   txt_save(file)
   if backup_enabled:
      zip(file)
```

Stratégie de tolérance aux erreurs

Types d'erreurs

- audio corrompu
- crash modèle
- conflit mémoire
- I/O

Stratégie

- try/catch individuel,
- logs détaillés,
- retry automatique (max 3).

Pseudo-code

```
for f in files:
  for attempt in range(3):
    try:
      process(f)
      break
    except:
      log("error")
  else:
    log("failed permanently")
```

Gestion des données et formats

Vue générale

Le système Station TV manipule quatre catégories principales de données :

1. **Données audio source** : fichiers MP3 issus des chaînes TNT.
2. **Données intermédiaires** : WAV normalisés, segments internes, métriques QoS.
3. **Données finales** : transcriptions et exports multi-formats.
4. **Données de support** : logs, configurations, rapports, backups.

La gestion des données est organisée selon trois principes clés :

- **Standardisation des formats** pour compatibilité Whisper.
- **Structuration riche** pour exploitation aval.
- **Sécurité et traçabilité** pour opérations longues et critiques.

L'ensemble des données est traité de manière **incrémentale** pour limiter les risques de perte en cas de crash.

Formats d'entrée

Fichiers audio MP3

Propriété	Valeur
Encodage	MP3
Fréquence	48 kHz
Débit	256 kbps
Mono	Variable (mono/dual)

Métadonnées extraites automatiquement :

- nom du fichier
- durée audio
- chaîne
- horaire diffusion

Structure CSV d'inventaire générée :

filename,channel,start_time,duration_s,path
tf1_03112025_2030.mp3,TF1,2025-11-03 20:30,312,/data/input/tf1_0311.mp3

Données intermédiaires

Fichiers WAV normalisés

Propriété	Valeur
Format	PCM16
Fréquence	48 kHz
Canaux	Mono
Normalisation	Oui

Nom standardisé :

<original>_norm.wav

Segments internes JSON

Les segments transcrits par Whisper avant export sont stockés en JSON structuré.

Structure interne d'un segment :

```
{
 "start": 3.02,
 "end": 5.17,
 "text": "Bonsoir et bienvenue à tous",
 "confidence": 0.87,
 "speaker": null
}
```

Historique des métriques QoS

Fichiers CSV par run, append temps réel :

```
timestamp,cpu,ram,throughput,wer
2025-11-21 12:30:00,74.5,62.3,1.45,0.09
```

Formats de sortie

Le système produit quatre types de données finales.
Chaque export correspond à un use-case métier différent.

TXT (texte brut)

Objectif : lecture humaine, traitement linguistique simple.

Format :

```
[00:00:02] Bonsoir à tous et bienvenue.
[00:00:05] Ce soir dans notre émission...
```

Caractéristiques :

- UTF-8
- timestamps optionnels
- un fichier par source audio

JSON (structure riche)

Objectif : exploitation machine, APIs, intégration Station TV.

Structure générale :

```
{
  "file": "tf1_03112025_2030.mp3",
  "channel": "TF1",
  "duration": 312,
  "model": "medium",
  "segments": [...]
}
```

CSV (tableau consolidé multi-sources)

Objectif : analyse statistique, import Excel, data warehouse.

Exemple :

```
file,channel,start,end,text,confidence
tf1_x,TF1,3.02,5.17,"Bonsoir à tous...",0.87
```

Avantages :

- exploitation rapide
- agrégation simple
- format universel

SRT (sous-titres horodatés)

Objectif : vidéo, accessibilité, médias.

Exemple :

```
1
00:00:02,000 --> 00:00:05,170
Bonsoir à tous et bienvenue.

2
00:00:05,170 --> 00:00:07,800
Ce soir dans notre émission...
```

Contraintes :

- segment < 4 s recommandé
- texte lisible et fluide

Modèles de données internes

Représentation d'un fichier audio

```
class AudioFile:
    path: str
    duration: float
    channel: str
    timestamp: datetime
```

Représentation d'un segment transcrit

```
class Segment:
    start: float
    end: float
    text: str
    confidence: float
    speaker: Optional[str]
```

Données de support

Fichier de configuration YAML

Structure typique :

```
hardware:
  cpu_processes: 6

whisper:
  model: medium
  timestamps: true

export:
  formats:
    - txt
    - json
    - csv
    - srt

logging:
  level: INFO
```

Objectifs :

- centraliser les paramètres
- permettre le tuning rapide
- assurer la reproductibilité

Journaux d'exécution

Caractéristiques :

- logs horodatés
- par module
- multi-niveaux : DEBUG, INFO, ERROR

Exemple :

[2025-11-21 12:42:01] INFO: Started process #2 with 64000 MB RAM

Backups

Stratégies :

- zip par fichier traité
- flush immédiat
- export sur NAS optionnel

Nom standardisé :

backup_2025-11-21_1530.zip

Sécurité des données

Le système applique des mesures de sécurité minimisant la perte d'information :

1. **Export incrémental**
   Un fichier exporté = un fichier sauvegardé.
2. **Redondance optionnelle NAS**
3. **Protection contre crash**
   - try/catch par fichier
   - logs erreurs
   - skip non bloquant
4. **Nom standardisé + métadonnées**
5. **Partitionnement par run**

Conformité et traçabilité

Chaque transcription est associée à des métadonnées permettant :

- la reconstitution du contexte
- la comparaison entre runs
- l'audit scientifique

Exemple JSON :

{
 "file": "france2_x",
 "date": "2025-11-21",
 "model": "medium",
 "ram": 6824,
 "cpu_avg": 74.12,
 "wer": 0.086
}

Tests unitaires

Objectifs des tests unitaires

Les tests unitaires du projet Station TV ont pour objectif de :

1. Vérifier le bon fonctionnement des modules critiques (Preprocessing, Core, QoS, Export, Utilities)
2. Détecter précocement les régressions
3. Garantir la stabilité du système dans un contexte de développement Agile
4. Fournir une base solide pour l'intégration continue
5. Sécuriser la montée en charge ultérieure

La priorité est donnée :

- aux composants à forte complexité algorithmique,
- aux fonctions manipulant des données critiques,
- aux fonctions exposées aux erreurs d'E/S.

Environnement de test

Langage et Framework

Technologie	Version
Python	3.10+
pytest	≥ 9.0

Organisation

Les tests sont regroupés dans le répertoire :

tests/

Le lancement se fait via :

python scripts/RunTests.py

ou directement :

pytest -v

Stratégie de test

La stratégie adoptée repose sur trois niveaux :

Tests unitaires

- Portent sur fonctions ou classes isolées
- Validation comportemental / structurel

Tests fonctionnels

- Portent sur modules complets
- Vérification cohérence I/O

Tests de non-régression

- Assurent la stabilité suite à modifications
- Tests déclenchés avant merge Git

Modules testés

Module Preprocessing

Fonctionnalités testées :

- conversion MP3 → WAV
- extraction de métadonnées
- segmentation optionnelle
- traitement fichiers invalides

Exemples de tests :

```
def test_convert_mp3_to_wav(tmp_path):
   wav = convert_mp3_to_wav("test.mp3", tmp_path/"test.wav")
   assert wav.exists()
```

Critères :

- durée correcte
- absence d'erreur

Module Core Transcription

Fonctionnalités testées :

- chargement de modèle Whisper
- transcription courte
- gestion multiprocess
- erreur fichier corrompu

Exemples :

```
def test_transcribe_short_clip():
   text = transcribe_file("short.wav")
   assert len(text) > 0
```

Critères :

- pas de crash
- sortie non vide
- timestamps ordonnés

Module QoS

Fonctionnalités testées :

- mesure CPU/RAM
- calcul throughput
- calcul WER
- export CSV

Exemple :

```
def test_compute_throughput():
    result = compute_throughput(600, 300)
    assert result == 2.0
```

Module Export

Fonctionnalités testées :

- génération TXT
- génération JSON
- cohérence des timestamps
- formats valides

Critères :

- export lisible
- absence de caractères invalides
- JSON parsable

Module Utilities

Fonctions testées :

- logging multi-niveaux
- parsing CSV
- lecture config YAML

Exemple :

```
def test_load_config():
    config = load_config("default.yaml")
    assert config["hardware"]["cpu_processes"] > 0
```

Critères de réussite

Un test unitaire est considéré comme réussi si :

- la fonction retourne un résultat attendu,
- aucun crash n'est observé,
- les données générées sont valides.

Un run complet est considéré valide si :

1. Tous les tests passent, ou
2. Les échecs sont documentés et justifiés.

Gestion des erreurs en test

Stratégies :

- tests spécifiques pour cas extrêmes
- fixtures pour données invalides
- simulations d'erreurs E/S

Exemple :

```
def test_corrupted_file():
   with pytest.raises(Exception):
      transcribe_file("null.wav")
```

Processus d'exécution

Le lancement des tests se fait idéalement :

- Avant chaque merge Git
- Avant une exécution batch longue
- Après chaque modification critique

Script de test automatisé :

```
python scripts/RunTests.py
```

Sortie attendue :

```
15 passed, 0 failed
```

Reporting

Les résultats des tests peuvent être :

- affichés en console,
- exportés en fichier texte,
- archivés dans Git.

Format type :

[TEST] test_convert_mp3_to_wav ..... OK
[TEST] test_transcribe_short_clip ... OK

Intégration avec pipeline Agile

Chaque sprint intègre :

- objectifs de tests spécifiques
- critères de validation
- rapport de tests

Gestion des erreurs et sécurité

Objectifs généraux

Le système Station TV est conçu pour fonctionner sur de longues périodes (plusieurs dizaines d'heures) et traiter un grand volume de données audio.
Dans ce contexte, la gestion des erreurs et la sécurité des données sont des enjeux majeurs.

Les objectifs sont :

1. **Assurer la continuité d'exécution**, malgré des erreurs locales.
2. **Éviter les crashs critiques**, notamment liés à la mémoire ou aux E/S.
3. **Garantir l'intégrité et la traçabilité des données produites**.
4. **Limiter les pertes de données en cas d'interruption soudaine**.
5. **Fournir suffisamment d'informations pour diagnostic et audit.**

Sources de risques identifiées

Les risques potentiels sont regroupés en cinq catégories :

Erreurs liées aux fichiers audio

- fichiers corrompus ou incomplets
- encodage non compatible
- durée non calculable

Erreurs logicielles

- exceptions Python non gérées
- dysfonctionnement librairie audio
- crash modèle Whisper

Erreurs système

- manque de RAM
- surcharge CPU
- saturation disque

Erreurs E/S

- permissions insuffisantes
- fichier temporaire inaccessible
- chemins invalides

Erreurs liées au monitoring

- métriques incohérentes
- échec génération graphique
- formats CSV invalides

Stratégie générale de tolérance aux erreurs

Le système adopte une politique de **résilience locale** :

Principe clé

Une erreur sur un fichier ne doit pas arrêter l'ensemble du pipeline.

Chaque unité de traitement (fichier audio) est gérée de façon isolée, avec :

- protection par try/catch,
- logs détaillés,
- tentatives multiples,
- skip contrôlé.

Illustration simplifiée

```
for f in files:
  try_three_times(process_file, f)
```

Résultat :

- un crash local ≠ crash global
- robustesse aux données hétérogènes
- pipeline long sécurisé

Mécanisme de retry automatique

Pour les opérations critiques (transcription, conversion), le système effectue jusqu'à **trois tentatives**.

Pseudo-code :

```
def try_three_times(func, arg):
  for attempt in range(3):
    try:
      return func(arg)
    except Exception as e:
      log_error(e, attempt)
  mark_as_failed(arg)
```

Avantages :

- réduction des erreurs transitoires
- meilleure stabilité sur longues sessions

Isolation mémoire et sécurité RAM

Pour éviter les crashs liés à la mémoire, le système met en œuvre :

1. **Processus indépendants**
   - pas de partage de modèle
   - allocation RAM isolée
2. **Dimensionnement dynamique**
   - limite auto du nombre de processus
3. **Monitoring RAM périodique**
4. **Arrêt contrôlé en cas de saturation**

Exemple de logique :

```
if ram_usage() > threshold:
   pause_or_kill_process()
```

Effets :

- crash RAM évité
- stabilité longue durée

Sécurité des données

Sauvegarde incrémentale

Chaque fichier transcrit est sauvegardé immédiatement.

Conséquences positives :

- perte maximale limitée à 1 fichier
- reprise facile après incident

Systèmes de backup

Option de backup automatique :

- ZIP par session
- export hors dossier de travail

Objectif :

Tolérance à une corruption totale du dossier local.

Format de données robuste

- UTF-8
- JSON validé
- CSV structuré
- SRT conforme à la norme

Gestion des erreurs système

Détection d'anomalies

Surveillance :

- CPU > X%
- RAM > Y%
- throughput trop faible
- erreurs répétées

Actions correctives

- arrêt d'un processus
- réduction du nombre de workers
- écriture d'un rapport d'erreur
- reprise partielle

Exemple

```
if ram_usage() > limit:
   decrease_process_count()
```

Logging et auditabilité

Les logs sont produits automatiquement, avec :

- timestamp
- niveau (INFO, WARNING, ERROR)
- contexte (module, fichier)
- message détaillé

Exemple :

[2025-11-21 12:32:15] ERROR: Transcription failed on file 'f2.wav', attempt 2

Objectifs :

- permettre diagnostic post-mortem
- analyser la performance du pipeline
- assurer traçabilité scientifique

Sécurité opérationnelle

Le script RunPipeline intègre :

- validation de configuration avant exécution
- vérification des chemins d'entrée / sortie
- estimation RAM vs modèle sélectionné
- avertissement si risque de crash

Exemple de protection :

```
if required_ram > available_ram:
   warn_user()
   abort()
```

Effet :

- erreurs préventives
- réduction des mauvaises configurations

Résilience du pipeline aux interruptions

Cas typiques :

- crash processus
- arrêt machine
- reboot imprévu

Grâce à :

- sauvegarde incrémentale
- logs détaillés
- fichiers marqués comme "done"

Le système peut :

- reprendre après crash,
- éviter tout retraitement inutile.

Robustesse globale

Grâce à l'ensemble de ces mécanismes, Station TV est capable de :

- traiter des volumes massifs,
- sur de longues périodes,
- avec un taux d'échec marginal,
- et une perte d'information quasi nulle.

La résilience est une caractéristique structurelle du système, pas un patch tardif.

Déploiement et automatisation

Objectifs du déploiement

Le déploiement du système Station TV vise à fournir une installation :

1. reproductible
2. automatisée autant que possible
3. facilement configurable
4. robuste sur de longues exécutions

Le système a été conçu pour être installé sur des architectures similaires à la station Dell Precision 5820, mais reste compatible avec d'autres configurations, sous réserve d'ajustement des ressources.

Prérequis matériels

Recommandations minimales pour un usage intensif :

Composant	Spécification recommandée
CPU	16–24 threads
RAM	≥ 128 Go (Medium) / ≥ 200 Go (Large)
GPU	Aucun (CPU-only)
Stockage	SSD NVMe
OS	Ubuntu 22.04 ou Windows Server 2022

Cette configuration a été sélectionnée pour répondre aux contraintes RAM/CPU élevées du modèle Whisper.

Prérequis logiciels

Environnement Python

- Python 3.10+
- pip (package manager)
- virtualenv (optionnel)

Bibliothèques nécessaires

Installées via :

```
pip install -r requirements.txt
```

Logiciel tiers

- FFmpeg (conversion audio)

Vérification :

```
ffmpeg -version
```

Installation

Clonage du projet

```
git clone <repository-url>
cd stationtv
```

Environnement virtuel

Linux :

```
python -m venv venv
source venv/bin/activate
```

Windows :

```
python -m venv venv
venv\Scripts\activate
```

Installation des dépendances

```
pip install -r requirements.txt
```

Configuration

Le comportement du système est piloté par un fichier YAML, typiquement :

```
hardware:
  cpu_processes: 6

whisper:
  model: medium
  timestamps: true

export:
  formats: [txt, json, srt, csv]

paths:
  input: "./data/input"
  output: "./data/output"
  logs: "./logs"
```

Points clés :

- nombre de processus ajusté automatiquement selon RAM
- choix du modèle (tiny → large)

Automatisation de l'exécution

Script d'exécution complet

L'exécution du pipeline complet est réalisée en un clic via :

```
python scripts/RunPipeline.py
```

Fonctions assurées automatiquement :

1. Scan répertoire audio
2. Prétraitement MP3 → WAV
3. Transcription multiprocess
4. Export multi-format
5. Monitoring QoS
6. Logging
7. Sauvegardes

Un fichier batch/PowerShell est fourni pour Windows :

```
RUN_PIPELINE.bat
```

Automatisation des tests

Les tests peuvent être lancés via :

```
python scripts/RunTests.py
```

ou :

```
pytest -v
```

Utilisation recommandée :

- avant chaque exécution longue
- avant merge Git
- avant modification configuration

Automatisation du monitoring

Le monitoring est intégré au run complet.

Rôle de l'automate QoS :

1. collecter métriques CPU / RAM
2. Aggréger throughput / WER
3. Exporter CSV + PNG
4. Calculer moyennes / variances
5. Signaler anomalies possibles

Structure standard des exports :

```
/output/
  monitoring_cpu.csv
  monitoring_memory.csv
  cpu_usage.png
  memory_usage.png
```

Gestion des erreurs automatisée

Pendant l'exécution, le système :

- détecte les erreurs locales
- applique retry automatique
- journalise l'événement
- skip si échec persistant

Aucune intervention manuelle nécessaire pour :

- erreurs temporaires
- fichiers corrompus
- saturation locale

Modes d'exécution

Mode batch complet

Pour traiter un grand lot de fichiers :

```
python scripts/RunPipeline.py
```

Mode unitaire

```
python scripts/BasicTestWhisper.py file.mp3
```

Optimisation du déploiement

Dimensionnement automatique des ressources

Le système évalue :

- RAM totale disponible
- RAM requise par modèle
- nombre optimal de processus

et ajuste dynamiquement.

Affinité CPU

- isolation des workers
- réduction des conflits
- amélioration throughput

Monitoring long terme

Adapté à des sessions > 5j :

- sérialisation CSV
- graphiques haute résolution
- analyse post-mortem

Sécurité opérationnelle

Protection avant exécution :

- validation configuration YAML
- vérification chemins d'entrée / sortie
- estimation RAM > seuil
- avertissement utilisateur

Protection pendant exécution :

- kill préventif
- prévention crash RAM
- suppression deadlocks

Reproductibilité

La reproductibilité est assurée par :

- environnement virtuel
- versionnement Git
- configuration externalisée YAML
- documentation exhaustive

Résultat :

Exécution identique sur n'importe quelle machine compatible.

Étapes complètes de mise en production

1. Installer dépendances
2. Configurer YAML
3. Lancer tests unitaires
4. Lancer pipeline
5. Surveiller QoS
6. Exporter résultats
7. Sauvegarde finale

Durée typique :

- quelques heures à plusieurs jours selon volume / modèle