Cahier des charges

NeoKodi

Version	1.0
Date de rendu	09 Fevrier 2026
Membres	Pier-Alexandre Rosa, Léo L'huillier
Établissement	EPTECH

1. PRÉSENTATION DU PROJET

1.1 Contexte général

Ce projet s'inscrit dans le cadre d'un cursus académique à Epitech visant à développer des compétences en ingénierie logicielle avancée. Il consiste à concevoir et développer un media center moderne inspiré de Kodi, en utilisant le langage Rust.

Kodi est aujourd'hui une référence dans le domaine des media centers open source, avec une base d'utilisateurs importante et un écosystème riche. Cependant, son architecture historique (C++) et sa complexité accumulée au fil des années présentent des opportunités d'amélioration en termes de performance, de sécurité mémoire et de maintenabilité.

Ce projet académique vise à explorer une approche moderne du développement d'un media center en exploitant les avantages du langage Rust (sécurité mémoire, performances, concurrence sans data races) tout en adoptant une méthodologie professionnelle complète.

1.2 Présentation de NeoKodi

NeoKodi est un media center moderne développé en Rust, conçu pour offrir une expérience utilisateur fluide et intuitive dans la gestion et la lecture de contenus multimédias. Le logiciel permet de :

- Organiser et indexer automatiquement des bibliothèques multimédias (vidéos, musiques, image)
- Lire différents formats de médias avec des contrôles avancés
- Enrichir automatiquement les contenus avec des métadonnées (affiches, résumés, notes)
- Étendre les fonctionnalités et ajouter des sources de contenus via un système de plugins

Le projet met l'accent sur une architecture modulaire et maintenable, facilitant les évolutions futures et l'apprentissage du code par de nouveaux contributeurs.

1.3 Public cible

Utilisateurs finaux primaires :

- Particuliers souhaitant gérer leur collection multimédia personnelle
- Niveau technique : utilisateur moyen à avancé, capable d'installer et configurer un logiciel

Utilisateurs secondaires :

- Développeurs souhaitant créer des extensions ou plugins
- Contributeurs open source intéressés par Rust et les applications multimédias
- Évaluateurs académiques et professionnels

Contextes d'usage :

- PC personnel (Windows)

1.4 Portée du projet

Ce que NeoKodi EST :

- Un media center fonctionnel avec les fonctionnalités essentielles
- Une démonstration de conception logicielle professionnelle en Rust
- Une base extensible pour de futures améliorations

Ce que NeoKodi N'EST PAS (pour la V1) :

- Un remplacement complet et exhaustif de Kodi
- Une solution commerciale destinée à la production immédiate
- Compatible avec l'écosystème de plugins Kodi existant
- Capable d’etre compiler et lancer sur macOs et Linux

---

2. OBJECTIFS ET ENJEUX

2.1 Objectifs pédagogiques

Apprentissage technique :

- Gérer un projet de développement d'envergure sur plusieurs mois
- Mettre en pratique les bonnes pratiques de développement (tests, CI/CD, documentation)
- Comprendre et implémenter une architecture logicielle complexe et modulaire
- Maîtriser le langage Rust et son écosystème (cargo, crates, ownership model)

Compétences transversales :

- Gestion de projet : planification, suivi, ajustements
- Travail en équipe : coordination, revue de code, communication
- Analyse de l'existant : étude de Kodi et de son architecture
- Prise de décision technique : Choix du language (voir annexe), choix de bibliothèques, patterns de conception

2.2 Objectifs techniques

Performance :

- Démarrage de l'application en moins de 3 secondes
- Lecture fluide de vidéos 1080p sans saccades (60 FPS minimum)
- Navigation réactive dans l'interface

Qualité logicielle :

- Couverture de tests > 70% du code critique
- Architecture modulaire permettant l'ajout de fonctionnalités sans refonte majeure
- Code documenté et maintenable (commentaires, documentation)
- Zero crash lors de l'utilisation normale (gestion robuste des erreurs)

Fonctionnalités essentielles :

- Lecture de formats vidéo courants (MP4, MKV, AVI) et audio (MP3, FLAC, OGG)

- Gestion d'une bibliothèque multimédia avec recherche
- Système de plugins fonctionnel

2.3 Enjeux du projet

Pour l'équipe :

- Démontrer la capacité à mener un projet complexe de A à Z
- Acquérir une expertise valorisable en Rust et en architecture logicielle
- Développer un portfolio project

Pour l'évaluation académique :

- Respect des deadlines et livrables attendus
- Qualité du code et de l'architecture
- Documentation technique
- Présentation professionnelle du projet

Techniques :

- Gérer la complexité inhérente au traitement multimédia temps réel
- Concevoir une API de plugins sécurisée et performante

---

3. PÉRIMÈTRE FONCTIONNEL

3.1 Fonctionnalités principales (MVP - Version 1.0)

3.1.1 Gestion de bibliothèque multimédia

Description :
Permet à l'utilisateur d'ajouter des dossiers contenant des fichiers multimédias qui seront automatiquement scannés, indexés et organisés.

Fonctionnalités détaillées :

- Ajout/suppression de sources (dossiers locaux)
- Scan automatique et détection de nouveaux fichiers
- Détection du type de média (film, image, audio)

- Rafraîchissement manuel ou automatique de la bibliothèque

Critères d'acceptation :

- L'utilisateur peut ajouter un dossier qui contient 100+ fichiers
- Le scan initial indexe tous les fichiers rapidement
- Les modifications (ajout/suppression de fichiers) sont détectées au prochain scan
- Les erreurs (fichiers corrompus, formats non supportés) sont loguées sans bloquer le scan

3.1.2 Lecture multimédia

Description :
Moteur de lecture capable de lire les formats vidéo et audio les plus courants avec des contrôles standard.

Formats supportés (V1) :

- Vidéo : MP4 (H.264), MKV (H.264), AVI
- Audio : MP3, FLAC, OGG Vorbis, AAC

Contrôles disponibles :

- Lecture / Pause / Stop
- Avance rapide / Retour rapide (±10 secondes)
- Contrôle du volume (0-100%)
- Barre de progression avec seek (clic pour aller à un moment précis)

Critères d'acceptation :

- Lecture fluide d'une vidéo 1080p à 60 FPS sans décrochage
- Les contrôles répondent rapidement
- Pas de fuite mémoire lors de lectures successives
- La position de lecture est sauvegardée pour reprendre plus tard

3.1.3 Enrichissement par métadonnées

Description :
Récupération automatique d'informations supplémentaires sur les médias (affiches, synopsis, note, casting) depuis des bases de données en ligne.

Sources de données (APIs) :

- MusicBrainz

Métadonnées récupérées :

- Musique : artiste

Critères d'acceptation :

- Les fichiers audio sont correctement identifiés automatiquement
- Les métadonnées sont récupérées en arrière-plan sans bloquer l'interface

3.1.4 Interface utilisateur

Description :
Interface graphique intuitive permettant de naviguer dans la bibliothèque et de contrôler la lecture.

Vues principales :

- **Vue Acceuil** : liste des type de media, parametres
- **Vue Bibliothèque** : liste/grille des médias
- **Vue Lecteur** : lecteur en plein écran avec contrôles overlay
- **Vue Recherche** : champ de recherche
- **Vue Paramètres** : configuration de l'application

Navigation :

- Navigation à la souris (clic, scroll, double-clic pour lire)

Critères d'acceptation :

- L'interface répond instantanément aux actions utilisateur
- La bibliothèque affiche de nombreux médias sans ralentissement
- Le design est cohérent et professionnel (charte graphique définie)

3.1.5 Gestion des playlists et files d'attente

Description :
Permet de créer des listes de lecture personnalisées et de gérer une file d'attente de lecture.

Fonctionnalités :

- Création/suppression/modification de playlists
- Ajout/retrait de médias dans une playlist
- File d'attente temporaire (ajout de médias à lire ensuite)
- Lecture aléatoire (shuffle) et répétition (repeat)
- Sauvegarde des playlists sur disque

Critères d'acceptation :

- Une playlist peut contenir 500+ éléments sans problème de performance
- Les playlists sont persistantes entre les sessions

3.1.6 Système de plugins et extensions

Description :
Architecture permettant d'étendre les fonctionnalités de NeoKodi via des plugins tiers.

Types de plugins supportés :

- **Scrapers** : sources alternatives de métadonnées

Architecture de plugins :

- Plugins écrits en Rust (compilation en bibliothèques dynamiques)
- API de plugin bien définie et stable

Critères d'acceptation :

- Au moins 1 plugins de démonstration fonctionnels (ex: scraper TMDb, streamer YouTube)
- Un plugin ne peut pas crasher l'application principale
- Documentation claire pour développer un plugin
- Interface dans l'application pour activer/désactiver des plugins

3.2 Hors périmètre (explicitement exclu de la V1)

- Sous-titres : SRT, ASS (si intégré dans le conteneur)
- Les images (affiches, pochettes) sont mises en cache localement
- Accessibilité : taille de police configurable, contrastes suffisants
- Sélection de pistes audio et sous-titres (si disponibles)
- Support du gamepad/télécommande (pour usage HTPC)
- L'ordre des médias peut être réorganisé par glisser-déposer
- Enregistrement de flux TV en direct (PVR)
- Accès distant à la bibliothèque via navigateur web
- Reprendre la lecture sur un autre appareil
- Suggestions basées sur l'historique de visionnage
- Accès à des partages SMB/NFS
- Récupération automatique de sous titres depuis OpenSubtitles
- Plusieurs utilisateurs avec leurs préférences
- Gestion de DRM ou contenu protégé
- Lecture de Blu-ray/DVD avec menus interactifs
- Compatibilité avec les plugins Kodi existants (Python)
- Application mobile native (Android/iOS)
- Streaming vers des appareils externes (Chromecast, AirPlay)

4. SPÉCIFICATIONS TECHNIQUES

4.1 Technologies et langages

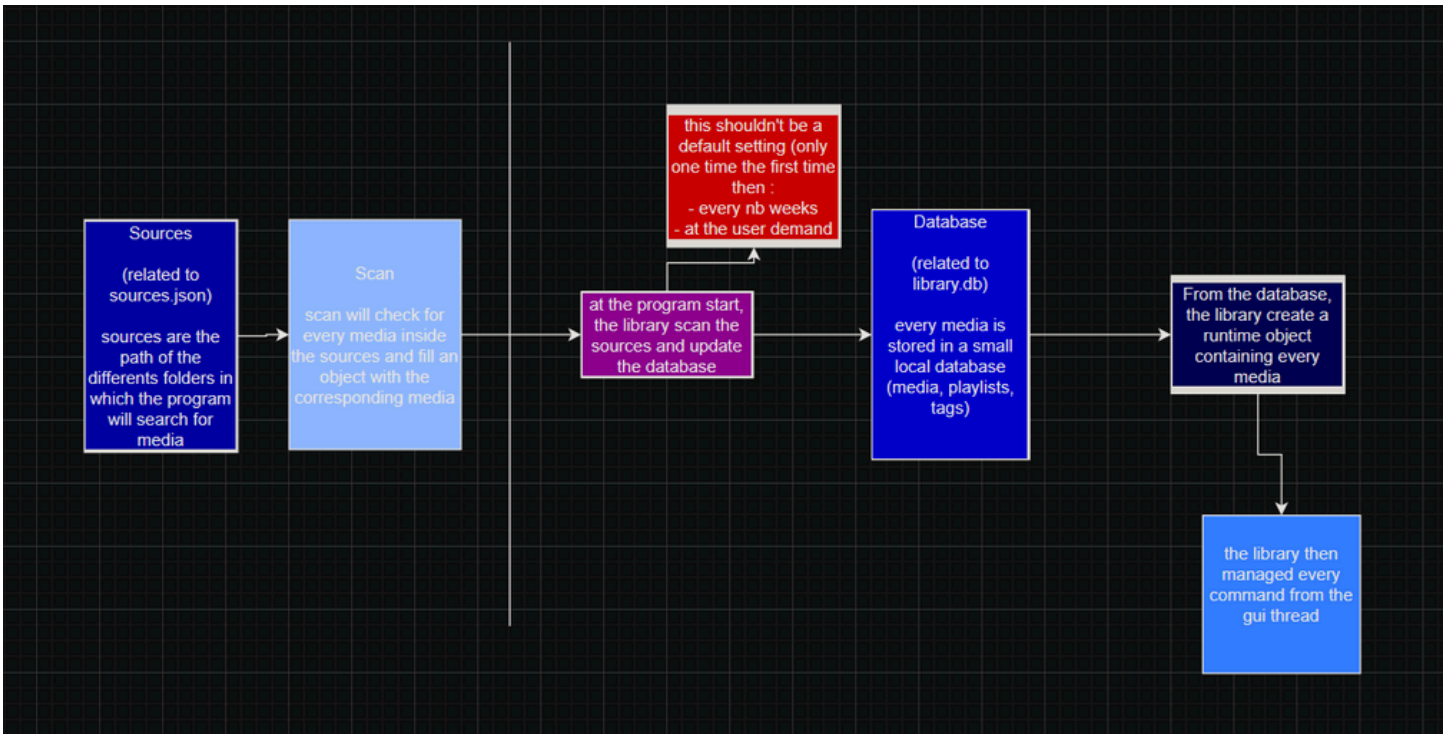
Langage principal :

- **Rust** (edition 2024, stable channel, voir annexe)
  - Justification : sécurité mémoire, performances, concurrence moderne
  - Courbe d'apprentissage reconnue et intégrée au planning

Bibliothèques principales (crates Rust) :

Domaine	Bibliothèque	Justification
Lecture vidéo/audio	base64 lofty = "0.22.4"	Prise en charge exhaustive des codecs
Interface graphique	dioxus dioxus-desktop	UI native cross-platform en Rust
Base de données	russqlite	SQL type-safe, embarqué, sans serveur
Parsing JSON	serde + serde_json	Sérialisation/désérialisation robuste
Tests	tokio-test	Tests async et mocking

4.2 Architecture logicielle



4.3 Modèle de données

Base de données : SQLite

-- Table des sources de médias

```
CREATE TABLE IF NOT EXISTS media (
    id INTEGER PRIMARY KEY,
    path TEXT UNIQUE NOT NULL,
    title TEXT,
    duration REAL,
    media_type TEXT
    status INTEGER DEFAULT 0,
    time_stop FLOAT DEFAULT 0.0

)
```

```
-- Tables des tags
        CREATE TABLE IF NOT EXISTS tags (
            id INTEGER PRIMARY KEY,
            name TEXT UNIQUE NOT NULL
        );


        CREATE TABLE IF NOT EXISTS media_tags (
            media_id INTEGER NOT NULL,
            tag_id INTEGER NOT NULL,
            PRIMARY KEY (media_id, tag_id),
            FOREIGN KEY (media_id) REFERENCES media(id) ON DELETE CASCADE,
            FOREIGN KEY (tag_id) REFERENCES tags(id) ON DELETE CASCADE
        );


-- Table des playlists
        CREATE TABLE IF NOT EXISTS playlists (
            id INTEGER PRIMARY KEY,
            name TEXT NOT NULL UNIQUE COLLATE NOCASE
        );


        CREATE TABLE IF NOT EXISTS playlist_items (
            playlist_id INTEGER NOT NULL,
```

```
    media_id INTEGER NOT NULL,
    PRIMARY KEY (playlist_id, media_id),
    FOREIGN KEY (playlist_id) REFERENCES playlists(id) ON DELETE CASCADE,
    FOREIGN KEY (media_id) REFERENCES media(id) ON DELETE CASCADE
);
```

Justifications du modèle :

- **SQLite** : léger, embarqué, sans configuration serveur, adapté pour application desktop
- **Normalisation partielle** : équilibre entre intégrité et performances
- **Genres en JSON** : flexibilité pour multiple genres par média
- **Index** : optimisation des requêtes fréquentes (recherche par titre, artiste)
- **Foreign keys avec CASCADE** : maintien automatique de l'intégrité référentielle

4.4 Plugins

4.4.1 Plugin api

L'application contient un dossier Plugin_api qui definit les fonctions que chaque plugin devra implementer

```
pub trait Plugin {
    fn name(&self) -> String;
    fn version(&self) -> String;
    fn plugin_type(&self) -> String; // "metadata",

    //get artist metadata by name
    fn metadata(&self, name: &str) -> String;
}
```

4.4.2 Add a new plugin

Pour créer un nouveau plugin, créer une lib rust qui implemente les fonctions de l’api (exemple ci-dessous), puis build la library et placer le binaire (dll, so, dylib) dans le dossier plugin ou directement ajouter le dans les parametres de l’application.

```
#[unsafe(no_mangle)]

pub extern "C" fn name() -> *mut c_char {
    let metadata = MusicBrainzMetadata;

    let name = metadata.name();

    CString::new(name).unwrap().into_raw()

}


#[unsafe(no_mangle)]

pub extern "C" fn version() -> *mut c_char {
    let metadata = MusicBrainzMetadata;

    let version = metadata.version();

    CString::new(version).unwrap().into_raw()

}


#[unsafe(no_mangle)]

pub extern "C" fn plugin_type() -> *mut c_char {
    let metadata = MusicBrainzMetadata;

    let plugin_type = metadata.plugin_type();

   CString::new(plugin_type).unwrap().into_raw()

}
```

---

5. ARCHITECTURE SYSTÈME

5.1 Vue d'ensemble de l'architecture

Modules :

- media
- database
- scan
- library
- gui
- logger
- threading
- plugin

5.2 execution détaillés du back end

Responsabilités :

- Décodage et lecture de fichiers audio/vidéo
- Gestion des flux (demuxing, decoding, rendering)
- Contrôles de lecture (play, pause, seek, volume)
- Gestion des pistes audio/sous-titres multiples
- Communication avec APIs externes (TMDb, MusicBrainz)

Interface publique :

```
pub trait Media: Send + Sync {

    fn init(&mut self);

    fn play(&mut self);

    fn pause(&self);

    fn resume(&self) ;

    fn stop(&self);
```

```
    fn info(&self) -> MediaInfo;

    fn media_type(&self) -> MediaType;

}
```

Threading :

- Thread dédié pour la gestion du back end et du front end
- Communication via channels pour contrôles et état

Plugins :

- Découverte de plugins dans le dossier plugins/
- Chargement dynamique et validation

5.3 Flux de données principaux

Flux 1 : Ajout d'une source et scan initial

[User] Ajoute un dossier

↓

[LibraryManager] Valide le chemin et crée une entrée dans DB

↓

[Scanner] Parcourt récursivement les fichiers

↓

[MediaDetector] Identifie le type (vidéo/audio) et extrait infos de base

↓

[Database] Insère les entrées media

↓

[MetadataScraper] (async) Récupère métadonnées pour chaque média

↓

[Database] Met à jour avec métadonnées enrichies

↓

[UI] Affiche la bibliothèque mise à jour

Flux 2 : Lecture d'un média

[User] Double-clic sur un média dans la bibliothèque

↓

[UI] Envoie événement PlayMedia(media_id)

↓

[PlaybackController] Récupère le chemin du fichier depuis DB

↓

[MediaEngine] Charge et démarre la lecture

↓

[Renderer] Affiche vidéo + audio

↓

[PlaybackController] Met à jour position de lecture toutes les 5 sec

↓

[Database] Sauvegarde position

Flux 3 : Installation d'un plugin todo : make it correct

[User] Télécharge un fichier ou creer une library de plugin

↓

[Plugin] Build la library

↓

[PluginManager] Check les plugins/nom_plugin

↓

[PluginManager] Charge les bibliothèques dynamique en verifiant le type de plugin

↓

[PluginManager] Appelle plugin.init()

↓

[UI] Affiche le plugin dans la liste des extensions

5.4 Gestion de la concurrence

Rust ownership model :
Rust garantit la sécurité mémoire à la compilation. Les problèmes de data races sont impossibles.

Approche multi-threading :

1. **Thread UI principal** :
   - Gère l'interface graphique (event loop)
   - Doit rester réactif (aucune opération bloquante)
   - Décodage et rendu audio/vidéo
2. **Thread Media Engine** :
   - lectures audios
   - Haute priorité pour éviter les saccades
   - Scan de fichiers
   - Requêtes réseau (métadonnées) via plugin
   - Accès base de données
   - Gestion des echanges avec le thread UI

Communication entre threads :

```
// Exemple : UI → Media Engine

use tokio::sync::mpsc;

enum PlayerCommand {

    Play,

    Pause,

    Seek(Duration),
```

```
    Stop,
}

// Dans UI
let (tx, rx) = mpsc::channel(100);
tx.send(PlayerCommand::Play).await?;

// Dans MediaEngine
while let Some(cmd) = rx.recv().await {
    match cmd {
        PlayerCommand::Play => self.play()?,
        PlayerCommand::Pause => self.pause()?,
        // ...
    }
}
```

Runtime asynchrone : Tokio

- Gestion efficace des I/O (réseau, fichiers)
- Évite le blocage du thread principal

Tests :

- CI/CD avec GitHub Actions : builds Linux, Windows
- Tests manuels sur machines physiques avant release

---

6. EXIGENCES NON-FONCTIONNELLES

6.1 Performance

Objectifs :

- **Démarrage rapide** : expérience utilisateur fluide, pas de frustration
- **UI réactive** : perception de rapidité (< 100ms = instantané pour l'humain)
- **60 FPS vidéo** : standard pour lecture fluide
- **Mémoire limitée** : compatible avec machines modestes (8GB RAM)

Stratégies d'optimisation :

- Indexation base de données pour recherches rapides
- Pooling de threads pour tâches parallèles

6.2 Fiabilité et disponibilité

Objectifs :

- **MTBF (Mean Time Between Failures)** : > 100 heures d'utilisation continue
- **Taux de crash** : < 1% des sessions utilisateur
- **Récupération automatique** : reprise après erreur sans perte de données

Gestion des erreurs :

```
// Pattern Rust : Result pour propagation d'erreurs

fn scan_library(path: &Path) -> Result<Vec<Media>, LibraryError> {

    // Toute erreur est explicite et gérée

    let files = list_files(path)?;

    let media = parse_media(files)?;

    Ok(media)

}

// Utilisation de log::error! pour tracer les problèmes
```

Résilience :

- Corruption de fichier : skip et log, continue le scan
- Plugin qui crash : isolation, désactivation automatique
- Base de données corrompue : rescan de la bibliotheque

Logging :

- Tous les événements critiques loggés
- Niveaux : ERROR, WARN, INFO, DEBUG

6.3 Sécurité

Menaces identifiées :

1. **Injection SQL**
   - Mitigation : requêtes paramétrées uniquement (sqlx avec type-safety)
2. **Path traversal**
   - Mitigation : validation stricte des chemins de fichiers
3. **Credentials dans logs**
   - Mitigation : sanitisation des logs (pas de mots de passe, tokens API)

Bonnes pratiques :

- Pas de unsafe Rust sauf absolument nécessaire
- Toutes les entrées utilisateur validées
- APIs externes appelées en HTTPS uniquement

6.4 Maintenabilité

Code quality :

- **Style** : respect de Rust conventions (rustfmt)
- **Tests** : couverture > 70% du code critique

Architecture maintenable :

- Séparation des responsabilités (SRP)
- Couplage faible entre modules
- Interfaces (traits) stables même si implémentation change
- Pas de dépendances circulaires

6.5 Utilisabilité (UX)

Principes de conception :

- **Simplicité** : fonctions essentielles visibles immédiatement
- **Feedback** : toute action utilisateur a une réponse visuelle (spinner, progress bar)
- **Tolérance aux erreurs** : confirmations pour actions destructives, undo si possible
- **Cohérence** : mêmes patterns d'interaction partout

Accessibilité :

- Contraste suffisant (WCAG AA minimum)

6.7 Évolutivité (Scalability)

Limites supportées (V2) :

- Bibliothèque : 50 000 médias
- Playlists : 10 000 médias par playlist
- Plugins : 50 plugins actifs simultanément

Performance à grande échelle :

- Virtualisation de la liste UI (affichage seulement des éléments visibles)
- Pagination des requêtes base de données
- Index optimisés pour grandes tables

---

7. GESTION DES RISQUES

7.1 Risques techniques

Risque 1 : Complexité de Rust pour débutants

Description :
Rust a une courbe d'apprentissage réputée difficile (ownership, lifetimes, borrow checker). L'équipe apprend le langage en parallèle du projet.

Probabilité : Haute
Impact : Élevé (retards, bugs, frustration)

Mitigation :

- Démarrer par des tutoriels Rust officiels (The Rust Book)
- Revue de code systématique entre membres de l'équipe
- Utiliser des patterns Rust bien documentés
- Ne pas hésiter à demander de l'aide (forums, Discord Rust)

Plan de contingence :

- Si blocage > 3 jours sur un concept Rust : simplifier l'approche
- Réduire le périmètre fonctionnel si nécessaire pour tenir les délais

Risque 2 : Choix de bibliothèque UI inadéquate

Description :
Les frameworks UI Rust sont encore jeunes (egui, iced, tauri). Risque de limitations découvertes tardivement.

Probabilité : Moyenne
Impact : Élevé (refonte UI nécessaire)

Mitigation :

- Benchmark comparatif dès octobre (prévu dans planning)
- Prototypes rapides avec chaque framework
- Critères de sélection clairs : performances, features, documentation, communauté

Plan de contingence :

- Si découverte de limitation majeure : switch vers un autre framework
- UI minimaliste en terminal (TUI avec ratatui) comme fallback extrême

Risque 3 : Problèmes de performance multimédia

Description :
Décodage vidéo temps réel complexe. Risque de saccades, latence, utilisation CPU excessive.

Probabilité : Moyenne
Impact : Critique (fonctionnalité principale compromise)

Mitigation :

- Utiliser FFmpeg (mature, optimisé, hardware acceleration)
- Tests de performance réguliers dès novembre
- Profiling pour identifier bottlenecks
- Hardware acceleration (VAAPI, NVDEC) si nécessaire

Plan de contingence :

- Si performances insuffisantes : limiter résolutions supportées (720p max)
- Alternativement : utiliser un lecteur externe (mpv, VLC) en subprocess

Risque 4 : Intégration de plugins complexe

Description :
Système de plugins dynamiques en Rust (unsafe, FFI) est avancé et risqué.

Probabilité : Haute
Impact : Moyen (fonctionnalité avancée, peut être réduite)

Mitigation :

- Étudier des exemples existants (cargo, rustc plugins)
- Utiliser libloading (crate éprouvée)

Plan de contingence :

- Si trop complexe : plugins en subprocess (IPC via stdin/stdout)
- Ou : plugins = scripts Python/Lua (interpréteur embarqué)
- Pire cas : pas de système de plugins en V1

7.2 Risques de planning

Risque 5 : Estimation trop optimiste des tâches

Description :
Estimation de 230h pour plugins + extensions semble très élevée et incertaine.

Probabilité : Haute
Impact : Élevé (dépassement deadline)

Mitigation :

- Réévaluation du planning à mi-parcours (fin novembre)
- Buffer de 20% intégré dans les estimations
- Priorisation stricte : MVP d'abord, nice-to-have après

Plan de contingence :

- Réduire le scope : moins de types de plugins supportés
- Reporter features secondaires (playlists avancées, configuration détaillée)
- Augmenter la charge de travail hebdomadaire si nécessaire (avec mesure)

Risque 6 : Disponibilité limitée de l'équipe

Description :
Équipe de 2 personnes, risque de maladie, autres projets, examens.

Probabilité : Moyenne
Impact : Moyen à élevé

Mitigation :

- Planning flexible avec tâches indépendantes
- Documentation continue pour reprendre le travail de l'autre
- Communication régulière (daily standup si possible)

Plan de contingence :

- Si un membre indisponible > 1 semaine : répartition urgente des tâches critiques
- Scope réduit si nécessaire pour tenir la deadline

7.3 Risques de qualité

Risque 7 : Manque de tests suffisants

Description :
Pression de temps peut pousser à négliger les tests.

Probabilité : Moyenne
Impact : Élevé (bugs en production)

Mitigation :

- TDD (Test-Driven Development) encouragé dès le début
- CI/CD bloque les merges si tests échouent ou couverture < 70%
- Temps de tests inclus dans estimation de chaque tâche

Plan de contingence :

- Sprint final dédié aux tests si nécessaire

Risque 8 : Problèmes de compatibilité multi-plateforme

Description :
Développement principalement sur un OS, découverte tardive de bugs sur autres OS.

Probabilité : Moyenne
Impact : Moyen

Mitigation :

- CI/CD teste sur Linux, Windows à chaque commit

Plan de contingence :

- Si bugs bloquants sur un OS : le marquer comme "support expérimental" pour V1

7.4 Tableau récapitulatif des risques

ID	Risque	Proba	Impact	Priorité	Mitigation clé
R1	Complexité Rust	Haute	Élevé	🔴 Critique	Formation intensive initiale
R2	Choix UI inadéquat	Moyenne	Élevé	🔴 Critique	Benchmark comparatif précoce
R3	Performance vidéo	Moyenne	Critique	🔴 Critique	
R4	Plugins complexes	Haute	Moyen	🟡 Élevée	Simplifier architecture
R5	Estimations optimistes	Haute	Élevé	🔴 Critique	Réévaluation mi-parcours
R6	Disponibilité équipe	Moyenne	Moyen	🟡 Élevée	Communication régulière
R7	Tests insuffisants	Moyenne	Élevé	🔴 Critique	CI/CD strict
R8	Bugs multi-plateforme	Moyenne	Moyen	🟡 Élevée	Tests automatisés CI

---

8. PLANNING ET LIVRABLES

8.1 Méthodologie de travail

Approche agile adaptée :

- **Sprints** : cycles de 2 semaines
- **Daily standups** : point rapide quotidien (10 min max)
- **Sprint planning** : début de chaque cycle, définir objectifs
- **Sprint review** : fin de cycle, démo des features terminées
- **Retrospective** : amélioration continue du process

Outils de gestion :

- **GitHub Project** : board Kanban (To Do, In Progress, Review, Done)
- **GitHub Issues** : chaque tâche = issue avec description, assigné, labels
- **Pull Requests** : revue de code systématique avant merge
- **Git workflow** : feature branches, pas de commit direct sur main

8.2 Planning détaillé avec livrables

	Octobre	Novembre	Decembre	Janvier
Preparation				
Project Setup & Foundations				
Core Media Engine				
User Interface				
Database & Library Management				
Advanced Features				
Plugin & Extension System				

8.3 Planning détaillé avec livrables

En tant qu'équipe de deux personnes, chaque membre sera impliqué dans tous les aspects du projet. Cependant, pour certaines tâches, il est prévu que l'un des deux membres prenne la direction, en fonction de ses forces et intérêts.

Le temps estimé pour chaque tâche inclut le développement des tests et de la documentation, ainsi que la résolution des éventuels bogues. Nous tenons également compte du fait que nous avons choisi d'implémenter le projet en Rust, un langage que nous sommes en train d'apprendre. Cela introduit naturellement une période d'adaptation et d'apprentissage initiale qui peut affecter notre productivité au début du projet.

Notre expérience limitée avec Rust rend également difficile l'estimation précise de la charge de travail pour certaines fonctionnalités, notamment la gestion des plugins, les systèmes d'extensions et l'intégration d'API externes. Nous prévoyons de réévaluer et d'affiner ces estimations une fois que nous aurons acquis une meilleure maîtrise du langage et obtenu des données concrètes sur l'avancement du développement.

Si le développement progresse plus rapidement que prévu, ce document sera mis à jour à la fin de chaque phase du projet afin de refléter la nouvelle planification et l'avancement.

Le temps estimé n'inclut pas les périodes de discussion ou de réflexion liées aux décisions de conception, les séances de brainstorming ou la coordination générale du projet.

Phase 1 : Préparation (19/10 - 25/10)

Objectifs :

- Comprendre l'existant (Kodi)
- Définir clairement le projet
- Mettre en place l'infrastructure

Tâches :

Tâche	Durée	Responsable	Livrable
Benchmark Kodi et alternatives	30h	PAR, LL	Document de benchmark (10 pages)
Setup GitHub repository	1h	LL	Repo public avec README
Setup GitHub Project	3h	LL	Board Kanban configuré
Rédaction cahier des charges	3h	LL	Ce document (version 1.0)

Critères de validation :

- ✅ Cahier des charges approuvé
- ✅ Repository accessible et structuré
- ✅ Équipe alignée sur la vision

Livrable principal : Cahier des charges complet + Repo initialisé

Phase 2 : Fondations du projet (26/10 - 07/11)

Objectifs :

- Structurer le projet Rust
- Mettre en place qualité et CI/CD
- Concevoir l'architecture détaillée

Tâches :

Tâche	Durée	Responsable	Livrable
Document de benchmark final	4h	PAR	Benchmark.md avec comparatif
Init projet Rust (workspace)	3h	LL	Cargo.toml, structure modules
Linting + formatting (clippy, rustfmt)	3h	LL	Config dans CI
CI/CD (GitHub Actions)	14h	LL	Pipelines build/test/lint
Architecture détaillée	60h	PAR, LL	Diagrammes + ADR documents

Critères de validation :

- ✅ cargo build fonctionne
- ✅ CI passe au vert
- ✅ Architecture validée et documentée

Livrable principal : Projet Rust fonctionnel avec CI/CD + Documentation d'architecture

Phase 3 : Moteur multimédia (08/11 - 21/11)

Objectifs :

- Implémenter le cœur de lecture audio/vidéo
- Scanner et indexer des fichiers

Tâches :

Tâche	Durée	Responsable	Livrable
Scan de dossiers (récursif)	10h	PAR	Module scanner avec tests
Lecture vidéo basique (FFmpeg)	10h	LL	Module player lecture MP4
Extraction métadonnées (durée, codec)	10h	PAR	Module metadata_extractor
Gestion de bibliothèque simple	10h	LL	CRUD médias en mémoire
Base de données SQLite	10h	PAR	Schéma DB + migrations
Bindings audio/vidéo Rust	10h	LL	Intégration FFmpeg
Contrôles de lecture (play/pause/seek)	10h	PAR	API contrôles fonctionnelle
État de lecture (position, durée)	10h	LL	State management backend

Critères de validation :

- ✅ CLI peut scanner un dossier et lister fichiers détectés
- ✅ CLI peut lire une vidéo MP4 (sans UI, juste audio)
- ✅ Base de données stocke médias correctement
- ✅ Tests unitaires couvrent > 70% du code

Livrable principal : Moteur média fonctionnel (backend) testable en CLI

Phase 4 : Interface utilisateur (22/11 - 05/12)

Objectifs :

- Créer une UI graphique basique mais fonctionnelle
- Connecter UI au backend média

Tâches :

Tâche	Durée	Responsable	Livrable
Fenêtre + liste de médias	16h	PAR	UI affiche bibliothèque
Contrôles de lecture (boutons)	16h	LL	Boutons play/pause/stop
Affichage métadonnées	16h	PAR	Infos film/musique affichées
Gestion états multiples	16h	LL	UI réactive aux changements
Connexion GUI ↔ Core	16h	PAR, LL	Communication UI/Backend

Critères de validation :

- ✅ L'application démarre avec une fenêtre
- ✅ La bibliothèque s'affiche avec métadonnées
- ✅ Clic sur un média lance la lecture
- ✅ Les contrôles fonctionnent

Livrable principal : Application GUI fonctionnelle (MVP visuel)

Phase 5 : Base de données et gestion bibliothèque (06/12 - 19/12)

Objectifs :

- Implémenter stockage persistant robuste
- Système de recherche et filtrage
- Organisation de la bibliothèque

Tâches :

Tâche	Durée	Responsable	Livrable
Stockage local complet	20h	PAR, LL	DB SQLite production-ready
Indexation + recherche	20h	LL	Recherche full-text rapide
Tags et catégorisation	20h	LL	Filtres par genre/année/etc
CRUD complet médias	20h	PAR, LL	Edit/delete médias dans UI

Critères de validation :

- ✅ Recherche retourne résultats en < 500ms sur 1000 entrées
- ✅ Filtres fonctionnent correctement
- ✅ Modifications persistantes entre sessions

Livrable principal : Bibliothèque complète avec recherche et persistance

Phase 6 : Fonctionnalités avancées (20/12 - 02/01)

Objectifs :

- Playlists et file d'attente
- Configuration utilisateur
- Répétition et lecture aléatoire

Tâches :

Tâche	Durée	Responsable	Livrable
Système de queue	2h	PAR	File d'attente temporaire
Repeat/Shuffle	2h	LL	Modes de lecture
Gestion playlists	2h	PAR	CRUD playlists
Configuration utilisateur	10h	LL	Settings UI + persistance

Critères de validation :

- ✅ Playlists créables et modifiables
- ✅ Shuffle/repeat fonctionnent
- ✅ Settings sauvegardés et appliqués

Livrable principal : Features de confort utilisateur

Phase 7 : Plugins et extensions (20/12 - 31/01)

Objectifs :

- Système de plugins fonctionnel
- Au moins 2 plugins de démonstration
- Documentation développeur

Tâches :

Tâche	Durée	Responsable	Livrable
Architecture plugins	60h	PAR, LL	API + loader
Plugin TMDb scraper	50h	PAR	Plugin fonctionnel
Plugin YouTube streamer	50h	LL	Plugin fonctionnel
Documentation plugins	30h	PAR, LL	Guide développeur
Intégration APIs externes	40h	PAR, LL	Clients HTTP + parsing

Critères de validation :

- ✅ 2 plugins fonctionnels et activables dans l'UI
- ✅ Documentation permet à un externe de créer un plugin
- ✅ Plugins ne peuvent pas crasher l'app

Livrable principal : Système extensible avec plugins de démo

Deadline

Project submission date	9 févr. 2026

Team

Name	Contact
Rosa Pier-Alexandre	pier-alexandre.rosa@epitech.eu
L'huillier Léo	leo.l-huillier@epitech.eu

Key resources and documents

- Github repository
- Github project

Annexe

Benchmark

1. Contexte du Projet

Ce projet vise à concevoir un logiciel multimédia multiplateforme moderne, inspiré de solutions existantes telles que Kodi, VLC ou Plex. L'application permettra de lire, organiser et diffuser des contenus audio et vidéo sur différents systèmes d'exploitation (Windows, Linux, macOS), tout en offrant une interface utilisateur moderne, fluide et entièrement personnalisable.
Au-delà des fonctionnalités de lecture basiques, le projet intègre également un système d'extensions modulaire permettant aux utilisateurs et développeurs tiers d'ajouter de nouvelles fonctionnalités, sources de contenu, ou intégrations avec des services externes (streaming, sous-titres, métadonnées, etc.).

1.1 Objectifs Techniques Principaux

Performance : Capacité à décoder et afficher des flux vidéo haute définition (4K, HDR) avec une utilisation optimale des ressources système (CPU, GPU, mémoire).

Fiabilité : Stabilité lors de lectures prolongées, gestion robuste des erreurs, et absence de fuites mémoire ou de plantages intempestifs.

Sécurité : Protection contre les vulnérabilités courantes (buffer overflows, race conditions), notamment lors du traitement de fichiers multimédias ou de l'exécution d'extensions tierces.

Extensibilité : Architecture permettant l'ajout facile de plugins, de codecs, ou de nouvelles interfaces utilisateur sans modifier le cœur de l'application.

Maintenabilité : Code lisible, bien structuré, et facile à faire évoluer sur le long terme par une équipe de développeurs.

Portabilité : Compilation native et déploiement simplifié sur Windows, Linux et macOS sans dépendances lourdes ou problèmes de compatibilité.

3. Analyse Détaillée des Langages

3.1 C++

Forces Principales :

Performances exceptionnelles : C++ offre un contrôle total sur la mémoire et les optimisations bas niveau. Les compilateurs modernes (GCC, Clang, MSVC) génèrent du code machine extrêmement optimisé, proche des performances théoriques du matériel. Pour un lecteur multimédia, cela se traduit par une utilisation minimale du CPU lors du décodage vidéo et une latence imperceptible.

Écosystème multimédia très mature : Quasi-totalité des bibliothèques multimédia de référence sont écrites en C/C++ : FFmpeg (décodage/encodage universel), libVLC, GStreamer, Qt Multimedia, SDL2, OpenAL. L'intégration est native et directe, sans overhead de FFI (Foreign

Function Interface).

Contrôle bas niveau : Accès direct au matériel (GPU via Vulkan/DirectX/Metal), optimisations SIMD manuelles (SSE, AVX), gestion précise de l'allocation mémoire pour minimiser les copies inutiles de buffers vidéo volumineux.

Portabilité éprouvée : Toolchains de compilation croisée bien établies, support sur toutes les plateformes imaginables (desktop, mobile, embedded).

Faiblesses Critiques :
Gestion mémoire manuelle dangereuse : Les bugs liés à la mémoire (fuites, double-free, dangling pointers, buffer overflows) sont la cause n°1 de vulnérabilités de sécurité dans les logiciels C++. Dans un lecteur multimédia manipulant des fichiers non fiables de sources variées, ces bugs peuvent conduire à des crashs fréquents ou pire, à des exploits de sécurité.
Complexité de compilation : Système de build complexe (CMake, Makefiles, etc.), temps de compilation longs sur les gros projets, gestion pénible des dépendances selon les OS.

Undefined Behavior (UB) : Le standard C++ regorge de comportements indéfinis (déréférencement de pointeur nul, overflow sur entiers signés, data races) qui rendent le debugging extrêmement difficile et peuvent produire des bugs non reproductibles.
Maintenance difficile : Code legacy souvent difficile à comprendre, refactoring risqué, absence de garanties du compilateur sur la validité des modifications.

Verdict : C++ reste techniquement solide pour des applications nécessitant des performances maximales, mais il impose un coût élevé en termes de développement, de debugging et de maintenance. Les risques de sécurité et de stabilité sont significatifs, surtout pour une application manipulant des médias de sources non fiables.

3.2 Python

Forces Principales :

Facilité de développement exceptionnelle : Syntaxe claire et concise, typage dynamique permettant un prototypage rapide, bibliothèque standard très riche.

Idéal pour le scripting et les extensions : Python excelle dans le rôle de langage de script. Il serait parfait pour un système de plugins permettant aux utilisateurs d'ajouter des fonctionnalités personnalisées (scrapers de métadonnées, intégrations avec des APIs de streaming, automatisations diverses).

Écosystème gigantesque : PyPI compte des centaines de milliers de packages. Pour les aspects non-critiques (parsing de fichiers de config, requêtes HTTP, manipulation de données structurées), Python dispose d'outils excellents (requests, BeautifulSoup, etc.).

Interopérabilité : Peut facilement interfacer avec du code C/C++ via ctypes, cffi ou Cython pour déléguer les parties critiques en performance.

Faiblesses Critiques :
Performances catastrophiques pour le runtime multimédia : L'interpréteur CPython est 50 à 100 fois plus lent que du code compilé natif. Pour décoder un flux vidéo 4K à 60fps, cela signifierait une utilisation CPU de 400-500% au lieu de 5-10% avec un langage natif. Totalement rédhibitoire pour le moteur principal.

Global Interpreter Lock (GIL) : Le GIL empêche l'exécution de bytecode Python en parallèle

sur plusieurs cœurs CPU. Pour un lecteur multimédia qui doit gérer simultanément le décodage vidéo, audio, le rendu UI, et les E/S réseau, cette limitation est dramatique.

Consommation mémoire élevée : Les objets Python ont un overhead mémoire important. Pour manipuler de larges buffers vidéo (plusieurs Mo par frame), cela devient problématique.

Distribution difficile : Créer un exécutable standalone nécessite des outils comme PyInstaller qui produisent des packages volumineux et fragiles. Dépendance à une runtime Python.

Verdict : Python n'est absolument pas adapté pour le cœur d'un lecteur multimédia, mais reste un excellent choix pour la couche d'extensions et de plugins. Une architecture hybride (moteur natif + API Python pour les extensions) serait envisageable.

3.3 C# / .NET

Forces Principales :
Excellent équilibre productivité/performance : C# est un langage moderne avec garbage collection automatique, ce qui élimine la majorité des bugs mémoire du C++, tout en offrant des performances bien supérieures à Python ou JavaScript (seulement 2-5x plus lent que du C++ optimisé selon les benchmarks).

Portabilité avec .NET (Core) : .NET 6+ est véritablement multiplateforme et open-source. Il est possible de compiler pour Windows, Linux et macOS depuis n'importe quelle plateforme.
.NET MAUI pour les UI natives : .NET MAUI permet de créer des interfaces utilisateur natives avec un code partagé, offrant de meilleures performances qu'Electron tout en conservant une expérience développeur agréable.

Syntaxe moderne et claire : LINQ, async/await, pattern matching, records… C# intègre beaucoup de fonctionnalités modernes qui rendent le code élégant et maintenable.

Tooling excellent : Visual Studio et Rider sont parmi les meilleurs IDEs disponibles, avec refactoring automatique, debugging puissant, et analyse statique intégrée.

Faiblesses Critiques :
Dépendance à la runtime .NET : Bien que .NET Core permette la compilation en binaire natif (avec NativeAOT), cela reste expérimental et limite certaines fonctionnalités (réflexion notamment). En mode standard, il faut distribuer la runtime .NET avec l'application.
Écosystème multimédia limité : Il n'existe pas d'équivalent .NET natif de FFmpeg ou GStreamer. Les solutions existantes (NAudio, LibVLCSharp) sont en réalité des wrappers autour de bibliothèques C/C++, ce qui réintroduit la complexité d'intégration.

Contrôle bas niveau limité : Bien que C# permette du code 'unsafe', il reste moins flexible que C++ ou Rust pour les optimisations fines. Les pauses GC (Garbage Collection) peuvent poser problème pour maintenir une latence constante dans le rendu vidéo.

Performances inférieures au natif : Même avec NativeAOT, les performances restent en deçà de langages sans GC comme C++ ou Rust. Pour un lecteur multimédia, chaque % d'utilisation CPU économisé compte.

Verdict : C# / .NET est un choix correct et pragmatique, particulièrement pour une équipe déjà familière avec l'écosystème Microsoft. Cependant, pour un projet open-source visant les meilleures performances possibles et un contrôle bas niveau maximal, ce n'est pas le choix optimal.

3.4 Rust

Forces Principales :

Performances au niveau du C++ : Rust génère du code natif aussi rapide que du C++ grâce à LLVM. Les benchmarks montrent des performances équivalentes, voire supérieures dans certains cas (grâce à des optimisations comme l'absence d'aliasing par défaut). Pour le décodage multimédia, cela signifie une utilisation CPU minimale et une fluidité maximale. Sécurité mémoire garantie par le compilateur : C'est LA killer feature de Rust. Le système de ownership et le borrow checker détectent à la compilation tous les bugs mémoire classiques : use-after-free, double-free, buffer overflows, dangling pointers. Cela élimine la cause n°1 de vulnérabilités de sécurité et de crashs dans les logiciels C/C++.

Multithreading sans data races : Le système de types de Rust garantit l'absence de data races à la compilation. On peut paralléliser agressivement le traitement multimédia (décodage audio/vidéo simultané, rendu, E/S) sans risquer de corruption de mémoire ou de comportements indéfinis. Le compilateur refuse simplement de compiler du code unsafe.

Écosystème multimédia en plein essor : Bien que plus jeune que celui du C++, l'écosystème Rust dispose déjà d'excellentes bibliothèques : gstreamer-rs (bindings officiels GStreamer), cpal (audio cross-platform), symphonia (décodage audio pur Rust), wgpu (API graphique moderne sur Vulkan/Metal/DirectX), ffmpeg-next (bindings FFmpeg). De nombreux projets majeurs (Firefox, Deno, Discord, Cloudflare) utilisent Rust pour les parties critiques.

Interopérabilité C/C++ excellente : Rust peut appeler du code C/C++ via FFI avec un overhead quasi nul. On peut donc réutiliser toutes les bibliothèques existantes (FFmpeg, libVLC) tout en écrivant le nouveau code de manière sûre en Rust.

Portabilité native : Cargo (le gestionnaire de paquets) et la compilation croisée fonctionnent remarquablement bien. Il est facile de produire des binaires pour Windows, Linux, et macOS depuis n'importe quelle plateforme. Pas de dépendance runtime externe.

Tauri pour les UI modernes : Tauri permet de créer des interfaces web (HTML/CSS/JS) avec un backend Rust, offrant le meilleur des deux mondes : UI moderne et backend performant. La consommation mémoire est minime comparé à Electron (10-20 Mo au lieu de 200+ Mo).
Slint pour les UI natives : Slint est un toolkit UI écrit en Rust, offrant des performances natives et un rendu GPU-acceleré. Idéal pour une interface 'big screen' type media center avec animations fluides.

Faiblesses :
Courbe d'apprentissage initiale : Rust a la réputation d'être difficile à apprendre. Les concepts de ownership, borrowing, et lifetimes demandent du temps pour être maîtrisés. Cependant, une fois ces concepts compris, ils deviennent une force (le compilateur devient un assistant qui empêche les bugs au lieu d'être un obstacle).

Last update 9 févr. 2026