

# **Documentation Complète de l'Agent d'Analyse Économétrique**

# 1. Architecture Globale

---

L'agent d'analyse économétrique est un système multi-agent composé de trois agents spécialisés qui travaillent ensemble pour réaliser une analyse économétrique complète à partir d'un fichier CSV et d'une requête utilisateur. L'architecture est conçue pour être modulaire, permettant à chaque agent de se concentrer sur une tâche spécifique dans le pipeline d'analyse.

Le système est orchestré par un script principal (`pipeline.py`) qui coordonne l'exécution séquentielle des agents. Chaque agent produit un résultat qui sert d'entrée à l'agent suivant, formant ainsi une chaîne de traitement cohérente.

Voici les composants principaux du système:

1. Pipeline (`pipeline.py`) - Orchestrateur qui gère l'exécution des agents  
2. Agent 1 (`agent1.py`) - Ingestion des données et problématisation académique  
3. Agent 2 (`agent2.py`) - Analyse économétrique et génération de visualisations  
4. Agent 3 (`agent3.py`) - Synthèse et génération du rapport final  
5. Utilitaires LLM (`llm_utils.py`) - Fonctions pour interagir avec les modèles de langage

Le système utilise des modèles de langage (LLMs) à travers deux backends possibles: - Ollama: pour l'exécution locale des modèles - Gemini: pour l'accès aux API de Google

Cette architecture permet de traiter des données économiques, de générer des analyses statistiques rigoureuses, et de produire des rapports académiques complets avec visualisations et interprétations.

## 2. Workflow et Flux de Données

---

Le workflow de l'agent d'analyse économétrique suit un flux séquentiel où chaque étape dépend des résultats de l'étape précédente:

1. L'utilisateur fournit un fichier CSV contenant des données économiques et une requête décrivant l'analyse souhaitée.
2. Le pipeline initialise l'environnement et orchestre l'exécution des agents.
3. L'Agent 1 (Ingestion et Problématisation): - Charge le fichier CSV et extrait des métadonnées détaillées - Détecte les problèmes potentiels dans les données - Conceptualise la question de recherche dans un cadre académique - Génère une introduction, une revue de littérature, des hypothèses et une méthodologie - Produit un fichier JSON contenant ces éléments
4. L'Agent 2 (Analyse Économétrique): - Utilise les résultats de l'Agent 1 comme contexte - Génère du code Python pour l'analyse statistique - Exécute ce code, en corrigeant automatiquement les erreurs si nécessaire - Produit des visualisations, des tableaux de régression et des analyses statistiques - Interprète les résultats avec l'aide du LLM - Génère un fichier JSON contenant les résultats et les interprétations
5. L'Agent 3 (Synthèse et Rapport): - Combine les résultats des Agents 1 et 2 - Génère une synthèse globale des résultats - Produit un document PDF bien structuré et formaté - Inclut toutes les visualisations avec leurs interprétations - Génère également un document Word si les bibliothèques requises sont disponibles
6. Le pipeline présente les résultats finaux à l'utilisateur et ouvre automatiquement le rapport PDF.

Flux de données: CSV + Requête Utilisateur → Agent 1 → JSON → Agent 2 → JSON → Agent 3 → PDF/DOCX

Chaque agent communique avec les modèles de langage via le module `llm_utils.py`, qui fournit une interface unifiée pour interagir avec différents backends (Ollama pour l'exécution locale, Gemini pour l'API cloud).

## 3. Détail des Composants

---

### 3.1 Pipeline Principal (pipeline.py)

Le pipeline.py est le script principal qui orchestre l'exécution de l'ensemble du système. Il prend en charge:

- Le parsing des arguments de ligne de commande
- La préparation de l'environnement (création des répertoires, initialisation des logs)
- L'exécution séquentielle des trois agents
- La gestion des erreurs et la communication entre les agents
- L'affichage des résultats finaux et l'ouverture automatique du rapport

Le pipeline garantit que chaque agent reçoit les entrées nécessaires et que les sorties sont correctement formatées pour l'agent suivant. Il maintient également des logs détaillés pour faciliter le débogage.

### 3.2 Agent 1: Ingestion et Problématisation (agent1.py)

L'Agent 1 est responsable de l'ingestion des données et de la problématisation académique. Ses fonctions principales sont:

- Lecture et analyse du fichier CSV pour extraire des métadonnées détaillées
- Détection des problèmes potentiels dans les données (valeurs manquantes, anomalies)
- Utilisation d'un LLM pour conceptualiser la question de recherche dans un cadre académique rigoureux
- Génération d'une introduction, d'une revue de littérature, d'hypothèses formelles et d'une méthodologie
- Analyse des limites méthodologiques et des variables clés
- Production d'un fichier JSON structuré contenant toutes ces informations

L'Agent 1 pose les fondements conceptuels de l'analyse, en transformant une requête utilisateur simple en un cadre de recherche académique rigoureux.

### 3.3 Agent 2: Analyse Économétrique (agent2.py)

L'Agent 2 est le cœur analytique du système, responsable de la génération et de l'exécution du code d'analyse économétrique. Ses fonctions principales sont:

- Génération de code Python pour l'analyse statistique et économétrique
- Exécution du code avec gestion automatique des erreurs et tentatives de correction
- Production de visualisations (graphiques, nuages de points, matrices de corrélation)
- Génération de modèles de régression et interprétation des résultats
- Capture et interprétation des sorties de visualisation
- Gestion robuste des erreurs avec tentatives multiples de correction, y compris l'utilisation de modèles plus puissants
- Production d'un fichier JSON contenant tous les résultats, visualisations et interprétations

L'Agent 2 est particulièrement sophistiqué, car il doit non seulement générer du code, mais aussi l'exécuter, capturer les erreurs, les corriger, et interpréter les résultats. Il inclut également des mécanismes de fallback vers des modèles plus puissants en cas d'échec répété.

### 3.4 Agent 3: Synthèse et Rapport (agent3.py)

L'Agent 3 est responsable de la synthèse et de la génération du rapport final. Ses fonctions principales sont:

- Agrégation des résultats des Agents 1 et 2 - Génération d'une synthèse globale des résultats - Création d'un raisonnement économique complet - Production de sections de discussion et de conclusion - Génération de références bibliographiques - Création d'un rapport PDF bien structuré et formaté - Production optionnelle d'un document Word si les bibliothèques requises sont disponibles

L'Agent 3 utilise des templates Jinja2 pour la génération de HTML, convertit ce HTML en PDF avec WeasyPrint, et peut également générer des documents Word avec python-docx. Il assure la présentation finale des résultats dans un format académique professionnel.

### 3.5 Utilitaires LLM (llm\_utils.py)

Le module llm\_utils.py fournit une interface unifiée pour interagir avec différents modèles de langage. Ses fonctions principales sont:

- Support de deux backends: Ollama (local) et Gemini (API cloud) - Gestion des appels aux modèles de langage avec ou sans images - Gestion robuste des erreurs et des timeouts - Parsing et validation des réponses - Logging détaillé pour faciliter le débogage

Ce module est utilisé par tous les agents pour communiquer avec les modèles de langage, offrant une couche d'abstraction qui simplifie l'interaction avec différents backends.

## 4. Technologies et Dépendances

---

L'agent d'analyse économétrique repose sur plusieurs technologies et bibliothèques clés:

Analyse de données et visualisation: - pandas: Pour la manipulation et l'analyse des données - matplotlib/seaborn: Pour la génération de visualisations - statsmodels: Pour les modèles économétriques et statistiques - numpy: Pour les calculs numériques

Modèles de langage: - Ollama: Interface pour les modèles de langage exécutés localement - API Gemini de Google: Pour l'accès aux modèles de langage dans le cloud

Génération de rapports: - Jinja2: Pour les templates HTML - WeasyPrint: Pour la conversion HTML vers PDF - python-docx (optionnel): Pour la génération de documents Word - Markdown: Pour le formatage du texte

Utilitaires et infrastructure: - logging: Pour la gestion des logs - argparse: Pour le parsing des arguments de ligne de commande - json: Pour la manipulation des données JSON - subprocess: Pour l'exécution de commandes externes - requests: Pour les appels API REST - base64: Pour l'encodage des images

Le système est conçu pour fonctionner dans un environnement Python 3.x et peut utiliser différents modèles de langage selon les besoins et la disponibilité.

## 5. Paramètres et Configuration

---

L'agent d'analyse économétrique peut être configuré via plusieurs paramètres:

Paramètres de ligne de commande (pipeline.py): - `csv_file`: Chemin vers le fichier CSV à analyser (obligatoire) - `user_prompt`: Requête utilisateur décrivant l'analyse souhaitée (obligatoire) - `--model`: Modèle LLM à utiliser (défaut: `gemma3:27b`) - `--backend`: Backend LLM à utiliser ('ollama' ou 'gemini', défaut: 'ollama') - `--academic`: Générer un rapport au format académique (activé par défaut)

Variables d'environnement: - `GEMINI_API_KEY`: Clé API pour l'accès à Gemini (obligatoire si `backend='gemini'`) - `GOOGLE_API_KEY`: Alias pour `GEMINI_API_KEY` (défini automatiquement) - `PYTHONUNBUFFERED`: Défini à "1" pour garantir un logging immédiat

Paramètres internes configurables: - Timeouts pour les appels API (`DEFAULT_GEMINI_TIMEOUT` dans `llm_utils.py`) - Modèles par défaut (`DEFAULT_OLLAMA_MODEL`, `DEFAULT_GEMINI_MODEL` dans `llm_utils.py`) - Nombre maximal de tentatives pour la correction de code (`max_attempts` dans `agent2.py`)

Le système utilise également des fichiers de log spécifiques pour chaque composant: - `pipeline.log`: Log principal du pipeline - `agent1.log`: Log de l'Agent 1 - `agent2.log`: Log de l'Agent 2 - `agent3.log`: Log de l'Agent 3

Ces logs peuvent être utilisés pour le débogage et le suivi de l'exécution.

## 6. Limitations et Extensions Possibles

---

### Limitations actuelles:

1. Dépendance aux modèles de langage: La qualité des analyses dépend fortement des capacités des modèles de langage utilisés.
2. Gestion des erreurs: Bien que le système tente de corriger automatiquement les erreurs, certaines situations complexes peuvent nécessiter une intervention manuelle.
3. Performance: L'exécution peut être lente, particulièrement avec des modèles locaux volumineux comme qwq:32b.
4. Validation des résultats: La validation des résultats économétriques reste limitée, et les interprétations peuvent parfois être trop générales.
5. Bibliothèques optionnelles: Certaines fonctionnalités (comme la génération de documents Word) dépendent de bibliothèques qui ne sont pas installées par défaut.

### Extensions possibles:

1. Support de formats de données supplémentaires: Ajouter la prise en charge d'autres formats comme Excel, parquet, ou SQL.
2. Interface utilisateur: Développer une interface web pour faciliter l'utilisation du système.
3. Parallélisation: Optimiser les performances en exécutant certaines tâches en parallèle.
4. Validation et tests automatisés: Ajouter des mécanismes de validation plus robustes pour les résultats économétriques.
5. Extensions multilingues: Ajouter le support pour d'autres langues que le français.
6. Modèles spécialisés: Intégrer des modèles de langage spécifiquement fine-tunés pour l'analyse économétrique.
7. Intégration avec des outils existants: Connecter le système à des environnements comme Jupyter, R Studio, ou des plateformes de visualisation comme Tableau.

## 7. Code Source Complet

---

Cette section présente le code source complet de chaque composant du système, avec des explications détaillées sur la structure et le fonctionnement du code.

## 7.1 Code Source: pipeline.py

Description:

Crée les répertoires nécessaires et nettoie ou fait pivoter les fichiers de log

Structure du fichier:

Fonctions:

- setup\_directories: Crée les répertoires nécessaires et nettoie ou fait pivoter les fichiers de log
- run\_agent: Exécute un agent et retourne son fichier de sortie JSON
- main: Pas de description

Code source complet:

```
#!/usr/bin/env python3
import argparse
import os
import subprocess
import sys
import json
import logging
from datetime import datetime
import shutil

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("pipeline.log"),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger("pipeline")

def setup_directories():
    """Crée les répertoires nécessaires et nettoie ou fait pivoter les fichiers de log"""
    # S'assurer que le répertoire outputs existe pour les logs et les sorties JSON/PDF
    os.makedirs("outputs", exist_ok=True)

    # Obtenir le chemin absolu du répertoire courant
    current_dir = os.path.abspath(os.getcwd())

    # Nettoyer ou faire pivoter les fichiers de log existants
    for log_file in ["agent1.log", "agent2.log", "agent3.log"]:
        log_path = os.path.join(current_dir, log_file)
        try:
            # Créer des fichiers vides
            with open(log_path, 'w') as f:
                pass
            logger.info(f"Fichier de log {log_path} vidé")

            # Vérifier les permissions
            os.chmod(log_path, 0o666) # Tous les droits de lecture/écriture
        except Exception as e:
            logger.warning(f"Problème avec le fichier de log {log_path}: {e}")

    logger.info("Répertoire 'outputs' créé et logs préparés")
```

```

def run_agent(agent_script, input_args, agent_name):
    """
    Exécute un agent et retourne son fichier de sortie JSON.
    """
    logger.info(f"Démarrage de l'agent: {agent_name}")

    # Vérifier si --model et --backend sont dans les arguments du pipeline
    if "--model" not in input_args:
        try:
            model_index = sys.argv.index("--model")
            model_value = sys.argv[model_index + 1]
            input_args.extend(["--model", model_value])
        except (ValueError, IndexError):
            logger.warning("Modèle non spécifié via --model, utilisation du défaut de l'agent")

    # Ajout du backend (gemini ou ollama)
    if "--backend" not in input_args:
        try:
            backend_index = sys.argv.index("--backend")
            backend_value = sys.argv[backend_index + 1]
            input_args.extend(["--backend", backend_value])
        except (ValueError, IndexError):
            logger.warning("Backend non spécifié via --backend, utilisation du défaut de l'agent")

    # Ajouter l'argument de log forcé pour chaque agent
    input_args.extend(["--log-file", f"{agent_name}.log"])

    logger.info(f"Commande pour {agent_name}: python3 {agent_script} {' '.join(input_args)}")

    try:
        # Obtenir le répertoire courant pour s'assurer que les chemins sont corrects
        current_dir = os.path.abspath(os.getcwd())

        # Créer un environnement qui force le logging immédiat
        env = os.environ.copy()
        env["PYTHONUNBUFFERED"] = "1"

        # Exécuter l'agent avec stdout capturé
        process = subprocess.Popen(
            ["python3", agent_script] + input_args,
            stdout=subprocess.PIPE,
            stderr=None, # Pas de capture de stderr - laisse le écrire directement
            text=True,
            env=env,
            cwd=current_dir
        )

        stdout, _ = process.communicate()

        if process.returncode != 0:
            logger.error(f"L'agent {agent_name} a échoué avec le code de retour {process.returncode}")
            sys.exit(f"Échec de l'agent {agent_name}")

        # Vérifier si la sortie est du JSON valide avant d'écrire
        try:
            json.loads(stdout)
        except json.JSONDecodeError as json_err:
            logger.error(f"La sortie de {agent_name} n'est pas du JSON valide: {json_err}")
            logger.error(f"Sortie reçue (premiers 500 chars):\n{stdout[:500]}")
            # Écrire quand même pour le débogage, mais avec une extension différente
            output_file = os.path.join("outputs", f"{agent_name}_output_invalid.txt")
            with open(output_file, "w", encoding="utf-8") as f:

```

```

        f.write(stdout)
        logger.error(f"Sortie brute enregistrée dans {output_file}")
        raise ValueError(f"Sortie invalide de {agent_name}") # Provoquer une erreur

    output_file = os.path.join("outputs", f"{agent_name}_output.json")
    with open(output_file, "w", encoding="utf-8") as f:
        f.write(stdout)

    logger.info(f"Agent {agent_name} terminé avec succès. Sortie enregistrée dans {output_file}")
    return output_file # Retourner le chemin du fichier JSON
except Exception as ex:
    logger.error(f"Erreur lors de l'exécution de {agent_name}: {ex}")
    sys.exit(f"Erreur pour {agent_name}: {ex}")

def main():
    parser = argparse.ArgumentParser(
        description="Pipeline académique multi-agent pour l'analyse économétrique"
    )
    parser.add_argument("csv_file", help="Chemin vers le fichier CSV")
    parser.add_argument("user_prompt", help="Prompt utilisateur décrivant l'analyse souhaitée")
    parser.add_argument("--model", default="gemma3:27b", help="Modèle LLM à utiliser (défaut: gemma3:27b)")
    parser.add_argument("--backend", default="ollama", choices=["ollama", "gemini"],
        help="Backend LLM à utiliser: 'ollama' (défaut) ou 'gemini'")
    parser.add_argument("--academic", action="store_true", default=True,
        help="Générer un rapport au format académique (activé par défaut)")
    args = parser.parse_args()

    # Conversion du chemin relatif en chemin absolu
    csv_file_absolute = os.path.abspath(args.csv_file)
    if not os.path.exists(csv_file_absolute):
        logger.error(f"Le fichier CSV spécifié n'existe pas : {csv_file_absolute}")
        sys.exit(1)
    logger.info(f"Chemin absolu du fichier CSV: {csv_file_absolute}")

    # Création des répertoires et préparation des logs
    setup_directories()

    # Exécution de l'agent 1 (Ingestion et problématisation académique)
    logger.info("=== DÉMARRAGE DE L'AGENT 1: INGESTION ET CONCEPTUALISATION ACADÉMIQUE ===")
    agent1_output_file = run_agent(
        "agent1.py",
        [csv_file_absolute, args.user_prompt, "--model", args.model, "--backend", args.backend],
        "agent1"
    )

    # Exécution de l'agent 2 (Analyse économétrique académique)
    logger.info("=== DÉMARRAGE DE L'AGENT 2: ANALYSE ÉCONOMÉTRIQUE ACADÉMIQUE ===")
    agent2_output_file = run_agent(
        "agent2.py",
        [csv_file_absolute, args.user_prompt, agent1_output_file, "--model", args.model, "--backend", args.backend],
        "agent2"
    )

    # Exécution de l'agent 3 (Synthèse académique et rapport)
    logger.info("=== DÉMARRAGE DE L'AGENT 3: SYNTHÈSE ACADÉMIQUE ET RAPPORT ===")
    agent3_output_file = run_agent(
        "agent3.py",
        # Passer les sorties des agents 1 et 2, et le prompt original
        [agent1_output_file, agent2_output_file, args.user_prompt, "--model", args.model, "--backend", args.backend],
        "agent3"
    )

    # Affichage du résultat final
    try:

```

```

with open(agent3_output_file, "r", encoding="utf-8") as f:
    result = json.load(f)

logger.info("=== PIPELINE ACADÉMIQUE TERMINÉE AVEC SUCCÈS ===")
print("\n" + "="*80)
print("RÉSUMÉ ACADÉMIQUE:")
print("="*80)
print(result.get("abstract", "Aucun résumé académique généré"))
print("\n" + "="*80)

if result.get("rapport_pdf"):
    print(f"Rapport académique complet généré: {result.get('rapport_pdf')}")
    # Ouvrir automatiquement le PDF si possible
    try:
        import platform

        system = platform.system()
        pdf_path = result.get('rapport_pdf')

        if system == 'Windows':
            os.startfile(pdf_path)
        elif system == 'Darwin': # macOS
            subprocess.call(['open', pdf_path])
        elif system == 'Linux':
            subprocess.call(['xdg-open', pdf_path])

        print("Le rapport PDF a été ouvert automatiquement.")
    except Exception as e:

        print(f"Le rapport a été généré mais n'a pas pu être ouvert automatiquement:")
    else:
        print("ERREUR : Le rapport PDF n'a pas pu être généré.")
        print(f"Détails de l'erreur : {result.get('error', 'Inconnus')}")

    print("="*80)

except FileNotFoundError:
    logger.error(f"Le fichier de sortie de l'agent 3 ({agent3_output_file}) est introuvable")
    sys.exit("Échec final du pipeline académique.")
except json.JSONDecodeError:
    logger.error(f"Le fichier de sortie de l'agent 3 ({agent3_output_file}) contient du texte non valide")
    sys.exit("Échec final du pipeline académique.")
except Exception as e:
    logger.error(f"Erreur lors de la lecture du résultat final: {e}")
    sys.exit("Échec final du pipeline académique.")

if __name__ == "__main__":
    main()

```

## 7.2 Code Source: agent1.py

Description:

Lit le fichier CSV et extrait les métadonnées détaillées

Structure du fichier:

Fonctions:

- generate\_metadata: Lit le fichier CSV et extrait les métadonnées détaillées
- detect\_data\_issues: Identifie les problèmes potentiels dans les données
- call\_llm\_for\_problematization: Appelle le LLM via le backend choisi pour problématiser les données de manière académique et rigoureuse
- parse\_llm\_output: Parse la sortie du LLM pour identifier les sections académiques
- main: Pas de description

Code source complet:

```
#!/usr/bin/env python3
import argparse
import json
import logging
import pandas as pd
import subprocess
import sys
import re

from llm_utils import call_llm

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("agent1.log"),
        logging.StreamHandler(sys.stderr)
    ]
)
logger = logging.getLogger("agent1")

def generate_metadata(csv_file):
    """
    Lit le fichier CSV et extrait les métadonnées détaillées
    """
    try:
        df = pd.read_csv(csv_file)
    except Exception as e:
        logger.error(f"Erreur lors de la lecture du fichier CSV: {e}")
        sys.exit(1)

    # Métadonnées basiques
    metadata = {
        "chemin_fichier": csv_file,
        "nb_lignes": len(df),
        "nb_colonnes": len(df.columns),
        "noms_colonnes": list(df.columns),
        "types_colonnes": {col: str(dtype) for col, dtype in df.dtypes.items()}
    }
```

```

# Statistiques par colonne
stats = {}
for col in df.columns:
    col_stats = {
        "valeurs_manquantes": int(df[col].isna().sum()),
        "pourcentage_manquant": float(round((df[col].isna().sum() / len(df)) * 100, 2))
    }

    # Statistiques numériques (si applicable)
    if pd.api.types.is_numeric_dtype(df[col]):
        col_stats.update({
            "min": float(df[col].min()) if not pd.isna(df[col].min()) else None,
            "max": float(df[col].max()) if not pd.isna(df[col].max()) else None,
            "moyenne": float(df[col].mean()) if not pd.isna(df[col].mean()) else None,
            "mediane": float(df[col].median()) if not pd.isna(df[col].median()) else None,
            "ecart_type": float(df[col].std()) if not pd.isna(df[col].std()) else None,
            "nb_valeurs_uniques": int(df[col].nunique())
        })
    else: # Statistiques catégorielles
        col_stats.update({
            "nb_valeurs_uniques": int(df[col].nunique()),
            "valeurs_frequentes": df[col].value_counts().head(5).to_dict() if df[col].nunique() > 5 else None
        })

    stats[col] = col_stats

metadata["statistiques"] = stats
return metadata

def detect_data_issues(metadata):
    """
    Identifie les problèmes potentiels dans les données
    """
    issues = []

    # Vérification des valeurs manquantes
    for col, stats in metadata["statistiques"].items():
        if stats["pourcentage_manquant"] > 0:
            issues.append({
                "type": "valeurs_manquantes",
                "colonne": col,
                "pourcentage": stats["pourcentage_manquant"],
                "description": f"La colonne '{col}' contient {stats['pourcentage_manquant']}% de valeurs manquantes."
            })

    # Détection des valeurs aberrantes pour les colonnes numériques
    for col, stats in metadata["statistiques"].items():
        if "ecart_type" in stats and stats["ecart_type"] is not None:
            if stats["ecart_type"] == 0:
                issues.append({
                    "type": "colonne_constante",
                    "colonne": col,
                    "description": f"La colonne '{col}' contient une valeur constante ({stats['valeurs_frequentes']})."
                })

    # Détection des colonnes à faible variabilité
    for col, stats in metadata["statistiques"].items():
        if stats["nb_valeurs_uniques"] == 1:
            issues.append({
                "type": "variabilite_nulle",
                "colonne": col,
                "description": f"La colonne '{col}' ne contient qu'une seule valeur unique."
            })

```

```

    })

    return issues

def call_llm_for_problematization(metadata, issues, user_prompt, model, backend):
    """
    Appelle le LLM via le backend choisi pour problématiser les données
    de manière académique et rigoureuse.
    """
    prompt = f"""## ANALYSE ÉCONOMÉTRIQUE ACADÉMIQUE - PHASE DE CONCEPTUALISATION

    ### Métadonnées du fichier
    ```json
    {json.dumps(metadata, indent=2, ensure_ascii=False)}
    ```

    ### Problèmes potentiels identifiés
    ```json
    {json.dumps(issues, indent=2, ensure_ascii=False)}
    ```

    ### Demande de l'utilisateur
    {user_prompt}

    ---

```

En tant qu'économètre et chercheur académique, vous êtes chargé de conceptualiser une étude

IMPORTANT : Votre réponse doit être ENTièrement EN FRANÇAIS, y compris tous les termes techniques

Veuillez produire:

1. **\*\*INTRODUCTION ACADÉMIQUE\*\***
  - Développez une introduction détaillée et approfondie sans limite de mots
  - Contextualisez largement le sujet dans la littérature économique
  - Présentez la problématique générale de manière exhaustive
  - Justifiez l'importance de cette question de recherche
  - Explorez les implications théoriques et empiriques du sujet
  - Assurez-vous que l'introduction soit complètement en français et très détaillée
2. **\*\*REVUE DE LITTÉRATURE SOMMAIRE\*\***
  - Mentionnez 5-7 références théoriques fondamentales (auteurs, théories) pertinentes pour
  - Identifiez les principaux mécanismes causaux étudiés dans la littérature
  - Situez cette recherche par rapport à la littérature existante
  - Explorez les débats académiques pertinents
3. **\*\*HYPOTHÈSES DE RECHERCHE FORMELLES (4-7 hypothèses)\*\***
  - Formulez des hypothèses testables précises sur les relations entre variables
  - Justifiez chaque hypothèse (fondement théorique)
  - Présentez-les sous forme H1, H2, H3...
  - Expliquez les mécanismes causaux sous-jacents
4. **\*\*MÉTHODOLOGIE PROPOSÉE\*\***
  - Spécification précise du/des modèle(s) économétrique(s) à estimer (équations)
  - Méthodes d'estimation appropriées (OLS, 2SLS, panel, etc.)
  - Tests de robustesse recommandés
  - Stratégies d'identification causale si pertinent
  - Justifications détaillées de vos choix méthodologiques
5. **\*\*LIMITES MÉTHODOLOGIQUES\*\***
  - Problèmes d'endogénéité potentiels
  - Biais de sélection ou d'auto-sélection
  - Problèmes de mesure ou d'erreurs
  - Propositions pour atténuer ces limites
  - Discussion des implications pour l'interprétation des résultats

## 6. \*\*VARIABLES CLÉS ET TRANSFORMATIONS\*\*

- Identifiez les variables dépendantes et indépendantes principales
- Proposez des transformations pertinentes (log, différences, interactions)
- Précisez la nature des variables (continues, catégorielles, etc.)
- Suggérez des variables instrumentales si nécessaire
- Discutez des problèmes potentiels de multicollinéarité

Votre réponse doit être structurée, précise et académique, intégrant la terminologie économé

"""

```
logger.info(f"Appel LLM via backend '{backend}' avec modèle '{model}'")
try:
    return call_llm(prompt=prompt, model_name=model, backend=backend)
except Exception as e:
    logger.error(f"Erreur lors de l'appel au LLM ({backend}): {e}")
    sys.exit(1)

def parse_llm_output(output):
    """
    Parse la sortie du LLM pour identifier les sections académiques
    """
    sections = {}

    # Recherche des sections par regex
    intro_pattern = r"(?:##\s*|)INTRODUCTION ACADÉMIQUE:?([\s\S]*?)(?==#\s*|REVUE DE LITTÉRA
    lit_review_pattern = r"(?:##\s*|)REVUE DE LITTÉRATURE:?([\s\S]*?)(?==#\s*|INTRODUCTION A
    hypotheses_pattern = r"(?:##\s*|)HYPOTHÈSES DE RECHERCHE:?([\s\S]*?)(?==#\s*|INTRODUCTIO
    methodology_pattern = r"(?:##\s*|)MÉTHODOLOGIE PROPOSÉE:?([\s\S]*?)(?==#\s*|INTRODUCTION
    limitations_pattern = r"(?:##\s*|)LIMITES MÉTHODOLOGIQUES:?([\s\S]*?)(?==#\s*|INTRODUCTI
    variables_pattern = r"(?:##\s*|)VARIABLES CLÉS:?([\s\S]*?)(?==#\s*|INTRODUCTION ACADÉMIQ

    # Extraction des sections académiques
    intro_match = re.search(intro_pattern, output, re.IGNORECASE)
    if intro_match:
        sections["introduction"] = intro_match.group(1).strip()

    lit_review_match = re.search(lit_review_pattern, output, re.IGNORECASE)
    if lit_review_match:
        sections["literature_review"] = lit_review_match.group(1).strip()

    hypotheses_match = re.search(hypotheses_pattern, output, re.IGNORECASE)
    if hypotheses_match:
        sections["hypotheses"] = hypotheses_match.group(1).strip()

    methodology_match = re.search(methodology_pattern, output, re.IGNORECASE)
    if methodology_match:
        sections["methodology"] = methodology_match.group(1).strip()

    limitations_match = re.search(limitations_pattern, output, re.IGNORECASE)
    if limitations_match:
        sections["limitations"] = limitations_match.group(1).strip()

    variables_match = re.search(variables_pattern, output, re.IGNORECASE)
    if variables_match:
        sections["variables"] = variables_match.group(1).strip()

    # Si aucune des sections académiques n'a été trouvée, essayer les patterns originaux
    if not sections:
        points_pattern = r"(?:##\s*|)POINTS DE VIGILANCE:?([\s\S]*?)(?==#\s*|PROBLÉMATISATIO
        problematisation_pattern = r"(?:##\s*|)PROBLÉMATISATION:?([\s\S]*?)(?==#\s*|POINTS D
        approches_pattern = r"(?:##\s*|)APPROCHES SUGGÉRÉES:?([\s\S]*?)(?==#\s*|POINTS DE VI

        points_match = re.search(points_pattern, output, re.IGNORECASE)
```

```

        if points_match:
            sections["points_vigilance"] = points_match.group(1).strip()

        problematisation_match = re.search(problematisation_pattern, output, re.IGNORECASE)
        if problematisation_match:
            sections["problematisation"] = problematisation_match.group(1).strip()

        approches_match = re.search(approches_pattern, output, re.IGNORECASE)
        if approches_match:
            sections["approches_suggerees"] = approches_match.group(1).strip()

    # Si les sections n'ont pas été trouvées, utiliser le texte entier
    if not sections:
        sections["output_complet"] = output.strip()

    return sections

def main():
    parser = argparse.ArgumentParser(
        description="Agent 1: Ingestion des données et problématisation"
    )
    parser.add_argument("csv_file", help="Chemin vers le fichier CSV")
    parser.add_argument("user_prompt", help="Prompt de l'utilisateur")
    parser.add_argument("--model", default="gemma3:27b", help="Modèle LLM à utiliser")
    parser.add_argument("--backend", default="ollama", help="Backend LLM: 'ollama' (local) o")
    parser.add_argument("--log-file", help="Fichier de log spécifique") # Nouveau paramètre
    args = parser.parse_args()

    # Reconfigurer le logging si un fichier de log spécifique est fourni
    if args.log_file:
        # Supprimer les handlers existants
        for handler in logger.handlers[:]:
            logger.removeHandler(handler)

        # Ajouter les nouveaux handlers
        file_handler = logging.FileHandler(args.log_file, mode='a') # mode 'a' pour append
        file_handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s'))
        logger.addHandler(file_handler)

        stream_handler = logging.StreamHandler(sys.stderr)
        stream_handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s'))
        logger.addHandler(stream_handler)

        logger.info(f"Logging redirigé vers {args.log_file}")

    # Génération des métadonnées
    logger.info(f"Génération des métadonnées pour {args.csv_file}")
    metadata = generate_metadata(args.csv_file)

    # Détection des problèmes
    logger.info("Détection des problèmes potentiels dans les données")
    issues = detect_data_issues(metadata)

    # Appel au LLM pour problématisation
    llm_output = call_llm_for_problematization(metadata, issues, args.user_prompt, args.model)

    # Parsing de la sortie
    logger.info("Parsing de la sortie du LLM")
    sections = parse_llm_output(llm_output)

    # Préparation de la sortie
    output = {
        "metadata": metadata,
        "data_issues": issues,

```

```
        "llm_output": sections,
        "raw_llm_output": llm_output,
        "user_prompt": args.user_prompt # Ajout du prompt utilisateur dans la sortie

    }

    # Sortie au format JSON
    print(json.dumps(output, ensure_ascii=False, indent=2))
    return 0

if __name__ == "__main__":
    main()
```

## 7.3 Code Source: agent2.py

Description:

Agent 2: Analyse Économétrique Ce script génère et exécute du code d'analyse économétrique à partir d'un fichier CSV et des suggestions fournies par l'agent 1. Il gère la correction automatique des erreurs, capture les visualisations et interprète les résultats. Usage: `python agent2.py chemin_csv prompt_utilisateur chemin_sortie_agent1 [--model modele] [--backend backend] [--auto-confirm]` Arguments: `chemin_csv`: Chemin vers le fichier CSV à analyser `prompt_utilisateur`: Prompt initial de l'utilisateur `chemin_sortie_agent1`: Chemin vers le fichier de sortie de l'agent 1 `--model`: Modèle LLM à utiliser (défaut: gemma3:27b) `--backend`: Backend LLM ('ollama' ou 'gemini', défaut: 'ollama') `--auto-confirm`: Ignorer la pause manuelle pour correction

Structure du fichier:

Classes:

- `CaptureOutput`: `def __init__`: Pas de description
- `NumpyEncoder`: Pas de description

Fonctions:

- `save_prompt_to_file`: Ajoute un prompt formaté au fichier journal spécifié
- `interpret_single_visualization_thread`: Fonction exécutée dans un thread pour interpréter une seule visualisation
- `capture_regression_outputs`: Capture les tables de régression OLS du texte de sortie et les enregistre à la fois comme texte et comme images
- `interpret_visualization_with_gemini`: Interprète une visualisation en envoyant directement l'image à Gemini via la fonction `call_llm`
- `generate_visualization_interpretations`: Génère des interprétations pour les visualisations SÉQUENTIELLEMENT (SANS THREADING)
- `__init__`: Pas de description
- `__enter__`: Pas de description
- `__exit__`: Pas de description
- `default`: Pas de description
- `_custom_show`: Pas de description
- `_custom_print`: Pas de description
- `save_figure`: Pas de description
- `interpret_regression_with_llm`: Interprète de manière détaillée les résultats d'une régression économétrique en utilisant un LLM
- `extract_regression_code`: Tente d'identifier le code qui a généré une régression spécifique
- `generate_analysis_code`: Génère le code d'analyse économétrique de niveau accessible Args: `csv_file`: Chemin absolu du fichier CSV `user_prompt`: Prompt initial de l'utilisateur `agent1_data`: Données de l'agent1 `model`: Modèle LLM à utiliser `prompts_log_path`: Chemin pour sauvegarder les prompts `backend`: Backend pour les appels LLM Returns: str: Code d'analyse généré

- `generate_analysis_narrative`: Fonction de placeholder pour la génération de narration explicative
- `main`: Fonction principale qui exécute le pipeline d'analyse économétrique

#### Code source complet:

```
#!/usr/bin/env python3
"""
Agent 2: Analyse Économétrique

Ce script génère et exécute du code d'analyse économétrique à partir d'un fichier CSV
et des suggestions fournies par l'agent 1. Il gère la correction automatique des erreurs,
capture les visualisations et interprète les résultats.

Usage:
    python agent2.py chemin_csv prompt_utilisateur chemin_sortie_agent1 [--model modele] [--

Arguments:
    chemin_csv: Chemin vers le fichier CSV à analyser
    prompt_utilisateur: Prompt initial de l'utilisateur
    chemin_sortie_agent1: Chemin vers le fichier de sortie de l'agent 1
    --model: Modèle LLM à utiliser (défaut: gemma3:27b)
    --backend: Backend LLM ('ollama' ou 'gemini', défaut: 'ollama')
    --auto-confirm: Ignorer la pause manuelle pour correction
"""

import argparse
import json
import logging
import os
import re
import sys
import subprocess
import tempfile
from datetime import datetime
import base64
import matplotlib.pyplot as plt
from io import StringIO
import pandas as pd
import threading # <--- AJOUTER CECI
import time      # <--- AJOUTER CECI

# Importation du module llm_utils
from llm_utils import call_llm

# =====
# Configuration du logging
# =====
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("agent2.log"),
        logging.StreamHandler(sys.stderr)
    ]
)
logger = logging.getLogger("agent2")

def save_prompt_to_file(prompt_content: str, log_file_path: str, prompt_type: str):
    """
    Ajoute un prompt formaté au fichier journal spécifié.
    Crée le répertoire si nécessaire.
```

```

Args:
    prompt_content: Contenu du prompt à sauvegarder
    log_file_path: Chemin du fichier journal
    prompt_type: Type de prompt (pour identification)
"""
try:
    log_dir = os.path.dirname(log_file_path)
    os.makedirs(log_dir, exist_ok=True)

    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    separator = "=" * 80

    with open(log_file_path, "a", encoding="utf-8") as f:
        f.write(f"{separator}\n")
        f.write(f"Timestamp: {timestamp}\n")
        f.write(f"Prompt Type: {prompt_type}\n")
        f.write(f"{separator}\n\n")
        f.write(prompt_content)
        f.write(f"\n\n{separator}\n\n")

    logger.info(f"Prompt '{prompt_type}' ajouté à {log_file_path}")

except Exception as e:
    logger.error(f"Impossible d'écrire le prompt '{prompt_type}' dans {log_file_path}: {e}")

def interpret_single_visualization_thread(vis, agent1_data, model, backend, prompts_log_path):
    """
    Fonction exécutée dans un thread pour interpréter une seule visualisation.
    Ajoute le résultat (ou une erreur) à la liste partagée 'results_list'.
    """
    # Utiliser le filename ou l'id comme identifiant unique
    if 'filename' in vis:
        vis_id = os.path.splitext(vis.get("filename", "unknown.png"))[0]
    else:
        vis_id = vis.get("id", f"unknown_{time.time()}") # Fallback ID unique

    try:
        logger.info(f"Thread démarré pour l'interprétation de: {vis_id}")
        interpretation = interpret_visualization_with_gemini(
            vis,
            agent1_data,
            model,
            backend,

            prompts_log_path
        )
        # Ajouter le résultat avec l'identifiant
        results_list.append({'id': vis_id, 'interpretation': interpretation})
        logger.info(f"Interprétation reçue pour {vis_id} dans le thread.")
    except Exception as e:
        logger.error(f"Erreur dans le thread pour {vis_id}: {e}")
        # Ajouter l'erreur avec l'identifiant
        results_list.append({'id': vis_id, 'interpretation': f"Erreur d'interprétation (thread)"})

# --- FIN NOUVELLE FONCTION ---

def extract_code(llm_output: str) -> str:
    """
    Extrait uniquement le code Python des blocs délimités par triple backticks.

    Args:
        llm_output: Texte complet de la sortie du LLM

    Returns:

```

```

        str: Code Python extrait ou texte complet si aucun bloc n'est trouvé
"""
code_blocks = re.findall(r"```(?:python)?\s*(.*?)```", llm_output, re.DOTALL)
if code_blocks:
    return "\n\n".join(code_blocks)
else:
    return llm_output

def remove_shell_commands(code: str) -> str:
    """
    Filtre le code en supprimant les lignes ressemblant à des commandes shell.

    Args:
        code: Code Python potentiellement contenant des commandes shell

    Returns:
        str: Code nettoyé
    """
    filtered_lines = []
    for line in code.splitlines():
        stripped = line.strip()
        if (stripped.startswith("pip install") or
            stripped.startswith("bash") or
            stripped.startswith("$") or
            (stripped.startswith("python") and not stripped.startswith("python3"))):
            continue
        filtered_lines.append(line)
    return "\n".join(filtered_lines)

def sanitize_code(code: str, csv_path: str, valid_columns: list[str]) -> str:
    """
    Force le bon chemin CSV, corrige les noms de colonnes, corrige les problèmes d'indentation
    et insère un bloc pour éviter les erreurs de corrélation.

    Args:
        code: Code Python à nettoyer
        csv_path: Chemin absolu du fichier CSV
        valid_columns: Liste des noms de colonnes valides

    Returns:
        str: Code Python nettoyé
    """
    import ast

    # Forcer le bon chemin CSV
    code = re.sub(
        r"pd\.read_csv\((.*?)\)",
        f"pd.read_csv('{csv_path}']",
        code
    )

    # Identifier les colonnes utilisées
    used_columns = set(re.findall(r"df\[['\"](.*?)['\"]\]", code)) | \
        set(re.findall(r"df\.([a-zA-Z_][a-zA-Z0-9_]*)", code))

    # Mapper les noms de colonnes utilisés aux noms réels
    col_map = {}
    for used in used_columns:
        for real in valid_columns:
            if used.lower() == real.lower():
                col_map[used] = real
                break

    # Corriger les noms de colonnes
    for used, correct in col_map.items():

```

```

        if used != correct:
            code = re.sub(rf"df\[['\"]{used}['\"]\]", f"df['{correct}']", code)
            code = re.sub(rf"df\.{used}\b", f"df['{correct}']", code)

# Ajouter un bloc pour éviter les erreurs de corrélation
if "df.corr()" in code or re.search(r"df\.corr\s*\(", code):
    init_numeric_block = "\n# ■ Sélection des colonnes numériques pour éviter les erreurs"
    match = re.search(r"(df\s*=\s*pd\.read_csv\(.*?\))", code)
    if match:
        insertion_point = match.end()
        code = code[:insertion_point] + init_numeric_block + code[insertion_point:]
    else:
        code = init_numeric_block + code

    code = re.sub(r"df\.corr\s*\(.*?\)", r"df_numeric.corr\1", code)

# NOUVEAU: Corriger les problèmes d'indentation avec plt.show() dans les blocs try/except
# Cette solution utilise une approche ligne par ligne pour une correction précise
lines = code.split('\n')
i = 0
while i < len(lines) - 1:
    current_line = lines[i].rstrip()
    next_line = lines[i + 1].rstrip()

    # Chercher plt.show() suivi par except, else ou elif à une indentation différente
    if current_line.strip() == 'plt.show()' and next_line.lstrip().startswith(('except',
        # Calculer l'indentation du bloc suivant
        next_indent = len(next_line) - len(next_line.lstrip())
        if next_indent > 0:
            # Réindenter plt.show() pour correspondre au bloc
            lines[i] = ' ' * next_indent + 'plt.show()'
    i += 1

# Reconstruire le code avec les lignes corrigées
code = '\n'.join(lines)

# MODIFICATION: Ne pas désactiver plt.show() complètement, mais le remplacer par une sau
code = re.sub(r"plt\.show\(\)", "plt.savefig('temp_figure.png', dpi=100, bbox_inches='ti

# Remplacer les styles Seaborn obsolètes
code = code.replace("seaborn-v0_8-whitegrid", "whitegrid")
code = code.replace("seaborn-whitegrid", "whitegrid")
code = code.replace("seaborn-v0_8-white", "white")
code = code.replace("seaborn-white", "white")
code = code.replace("seaborn-v0_8-darkgrid", "darkgrid")
code = code.replace("seaborn-darkgrid", "darkgrid")
code = code.replace("seaborn-v0_8-dark", "dark")
code = code.replace("seaborn-dark", "dark")
code = code.replace("seaborn-v0_8-paper", "ticks")
code = code.replace("seaborn-paper", "ticks")
code = code.replace("seaborn-v0_8-talk", "ticks")
code = code.replace("seaborn-talk", "ticks")
code = code.replace("seaborn-v0_8-poster", "ticks")
code = code.replace("seaborn-poster", "ticks")

return code

def capture_regression_outputs(output_text, output_dir, executed_code=""):
    """
    Capture les tables de régression OLS du texte de sortie et les enregistre
    à la fois comme texte et comme images. Tente également d'identifier le code
    qui a généré chaque régression.

```

Args:

output\_text: Texte de sortie contenant potentiellement des résultats de régression  
output\_dir: Répertoire où sauvegarder les résultats  
executed\_code: Code Python complet qui a été exécuté

Returns:

Liste des chemins des fichiers générés et des métadonnées associées  
"""

```
regression_outputs = []
```

```
regression_pattern = r"={10,}\s*\n\s*OLS Regression Results\s*\n={10,}(.*?)(?:\n={10,})\|\"
```

```
# Créer le répertoire de sortie s'il n'existe pas  
os.makedirs(output_dir, exist_ok=True)
```

```
# Chercher toutes les tables de régression dans la sortie  
regression_tables = re.findall(regression_pattern, output_text, re.DOTALL)
```

```
for i, table_content in enumerate(regression_tables):  
    table_id = f"regression_{i+1}"
```

```
    # Sauvegarder le contenu textuel  
    text_file_path = os.path.join(output_dir, f"{table_id}.txt")  
    with open(text_file_path, "w", encoding="utf-8") as f:  
        f.write(" "*80 + "\n")  
        f.write("OLS Regression Results\n")  
        f.write(" "*80 + "\n")  
        f.write(table_content)  
        f.write("\n" + " "*80 + "\n")
```

```
    # Créer une visualisation de la table et la sauvegarder comme image  
    img_file_path = os.path.join(output_dir, f"{table_id}.png")
```

```
    # Créer une visualisation de la table avec matplotlib  
    plt.figure(figsize=(12, 10))  
    plt.text(0.01, 0.99, "OLS Regression Results\n" + table_content,  
            family='monospace', fontsize=10,  
            verticalalignment='top')  
    plt.axis('off')  
    plt.tight_layout()  
    plt.savefig(img_file_path, dpi=100, bbox_inches='tight')  
    plt.close()
```

```
    # Encoder l'image en base64 pour l'inclure dans la sortie JSON  
    with open(img_file_path, 'rb') as img_file:  
        img_data = base64.b64encode(img_file.read()).decode('utf-8')
```

```
    # Extraire quelques métadonnées de base (R-squared, variables clés)  
    r_squared_match = re.search(r"R-squared:\s*([\d\.]+)", table_content)  
    r_squared = r_squared_match.group(1) if r_squared_match else "N/A"
```

```
    key_variables = re.findall(r"([A-Za-z0-9_])\s+[\d\.-]+\s+[\d\.-]+\s+[\d\.-]+\s+[\d\.-]"
```

```
    # Extraire quelques statistiques clés pour les données de régression
```

```
    regression_data = {  
        "r_squared": r_squared,  
        "variables": key_variables[:5], # Limiter aux 5 premières variables  
        "statistics": {}  
    }
```

```
    coef_section = re.search(r"==+\s*\n\s*(coef.*?)?:\n\s*==+|\Z)", output_text, re.DOTALL)  
    if coef_section:  
        lines = coef_section.group(1).split('\n')
```

```

headers = None
coef_data = []

for line in lines:
    if "coef" in line.lower():
        # C'est la ligne d'en-tête
        headers = re.split(r'\s{2,}', line.strip())
    elif line.strip() and headers:
        # C'est une ligne de données
        values = re.split(r'\s{2,}', line.strip())
        if len(values) >= len(headers):
            variable = values[0]
            coef_data.append({
                "variable": variable,
                "coef": values[1] if len(values) > 1 else "N/A",
                "std_err": values[2] if len(values) > 2 else "N/A",
                "p_value": values[4] if len(values) > 4 else "N/A"
            })

regression_data["coefficients"] = coef_data

# NOUVELLE SECTION: Convertir les données de régression en CSV amélioré pour faci
try:
    import pandas as pd
    if coef_data:
        coef_df = pd.DataFrame(coef_data)

        # Ajouter une colonne de significativité pour faciliter l'interprétation
        try:
            coef_df['significatif'] = coef_df['p_value'].astype(float) < 0.05
        except:
            # Si conversion en float échoue, on continue sans cette colonne
            pass

        # Ajouter des métadonnées contextuelles en commentaire
        csv_header = f"# Résultats de régression - R²: {r_squared}\n"
        csv_header += f"# Interprétation: un coefficient positif indique une rel
        csv_header += f"# Variables explicatives: {'', '.join(key_variables[:5])

        regression_data["csv_data"] = csv_header + coef_df.to_csv(index=False)

        # Sauvegarder en fichier CSV
        csv_path = os.path.join(output_dir, f"{table_id}_coefficients.csv")
        with open(csv_path, 'w', encoding='utf-8') as f:
            f.write(regression_data["csv_data"])
        logger.info(f"Données CSV améliorées de coefficients sauvegardées: {csv_
except Exception as e:
    logger.error(f"Erreur lors de la conversion des coefficients en CSV amélioré

# Extraire le code de régression si le code complet est fourni
regression_code = ""
if executed_code:
    regression_code = extract_regression_code(executed_code, table_content)
    regression_data["regression_code"] = regression_code

regression_outputs.append({
    'type': 'regression_table',
    'id': table_id,
    'text_path': text_file_path,
    'image_path': img_file_path,
    'base64': img_data,
    'metadata': {
        'r_squared': r_squared,
        'variables': key_variables[:5] # Limiter aux 5 premières variables
    },

```

```

        'data': regression_data, # Ajout des données structurées
        'csv_data': regression_data.get("csv_data", ""), # Ajout des données CSV améliorées
        'regression_code': regression_code # Ajout du code de régression
    })

    return regression_outputs

def interpret_visualization_with_gemini(vis, agent1_data, model, backend, prompts_log_path,
    """
    Interprète une visualisation en envoyant directement l'image à Gemini via la fonction call_llm

    Args:
        vis: Métadonnées de la visualisation avec base64 de l'image
        agent1_data: Données de l'agent1
        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM
        prompts_log_path: Chemin pour sauvegarder les prompts

    Returns:
        str: Interprétation de la visualisation
    """
    # Importer call_llm depuis llm_utils
    from llm_utils import call_llm

    # Valider que l'image est disponible
    if 'base64' not in vis or not vis['base64']:

        logger.error(f"Pas de données base64 disponibles pour la visualisation {vis.get('id', '')}")
        return "Erreur: Impossible d'analyser cette visualisation (image non disponible)"

    # Extraire le titre et le type
    if 'filename' in vis:
        filename = vis.get("filename", "figure.png")
        vis_id = os.path.splitext(filename)[0]
    else:
        vis_id = vis.get("id", "visualisation")
        filename = vis_id + '.png'

    # Déterminer le type de visualisation
    vis_type = "Unknown visualization"
    if 'regression' in vis_id.lower():
        vis_type = "Table de Régression OLS"
    elif 'correlation' in vis_id.lower() or 'corr' in vis_id.lower():
        vis_type = "Matrice de Corrélation"
    elif 'distribution' in vis_id.lower() or 'hist' in vis_id.lower():
        vis_type = "Graphique de Distribution"
    elif 'scatter' in vis_id.lower() or 'relation' in vis_id.lower():
        vis_type = "Nuage de Points"
    elif 'box' in vis_id.lower():
        vis_type = "Boîte à Moustaches"

    # Récupérer le titre
    title = vis.get("title", vis_type)

    # Extraire des informations supplémentaires pour le contexte
    metadata_str = json.dumps(agent1_data["metadata"], indent=2, ensure_ascii=False)
    extra_context = ""
    if 'metadata' in vis:
        if 'r_squared' in vis['metadata']:
            extra_context += f"\nR-squared: {vis['metadata']['r_squared']}"
        if 'variables' in vis['metadata']:
            extra_context += f"\nVariables principales: {' '.join(vis['metadata']['variables'])}"

```

```

        # Créer le prompt pour Gemini
        prompt = f"### INTERPRÉTATION DE VISUALISATION ÉCONOMIQUE

### Type de visualisation
{vis_type}

### Titre
{title}

### Identifiant
{vis_id}

### Métadonnées spécifiques
{extra_context}

### Contexte des données
Le dataset contient les variables suivantes: {' , '.join(agent1_data["metadata"].get("noms_co

### Métadonnées de l'ensemble de données
```json
{metadata_str[:500]}...
```

### Question de recherche initiale
{agent1_data.get("user_prompt", "Non disponible")}}

---

Analyse cette visualisation économique. Tu reçois directement l'image, donc base ton analyse

1. Décrire précisément ce que montre la visualisation (tendances, relations, valeurs aberran
2. Expliquer les relations entre les variables visibles
3. Mentionner les valeurs numériques spécifiques (minimums, maximums, moyennes) que tu peux
4. Relier cette visualisation à la question de recherche

Ton interprétation doit être factuelle, précise et basée uniquement sur ce que tu peux obser
"""

# Sauvegarder le prompt dans le fichier journal
save_prompt_to_file(prompt, prompts_log_path, f"Gemini Image Interpretation - {vis_id}")

try:
    # Utiliser call_llm de llm_utils qui gère maintenant les images
    logger.info(f"Appel à Gemini avec image pour visualisation {vis_id}")
    interpretation = call_llm(
        prompt=prompt,
        model_name=model,
        backend=backend,
        image_base64=vis['base64']
    )

    logger.info(f"Interprétation générée par Gemini pour: {vis_id}")
    return interpretation
except Exception as e:
    logger.error(f"Erreur lors de l'interprétation de l'image pour {vis_id}: {e}")
    return f"Erreur d'interprétation: {e}"

# --- REMPLACEMENT DE LA FONCTION ---
def generate_visualization_interpretations(visualizations, regression_outputs, agent1_data,
    """
    Génère des interprétations pour les visualisations SÉQUENTIELLEMENT (SANS THREADING).
    Utilise le LLM Gemini directement à partir des images.

    Args:

```

```

        visualizations (list): Liste des métadonnées des visualisations.
        regression_outputs (list): Liste des sorties des tables de régression.
        agent1_data (dict): Données de l'agent1.
        model (str): Modèle LLM à utiliser.
        backend (str): Backend pour les appels LLM.
        prompts_log_path (str): Chemin pour sauvegarder les prompts.
        timeout (int): Timeout en secondes pour chaque appel LLM d'interprétation. Défaut: 10

Returns:
    Liste mise à jour des visualisations avec interprétations.
"""
# Combiner visualisations et régressions
all_visuals = visualizations + regression_outputs
if not all_visuals:
    logger.info("Aucune visualisation ou table de régression à interpréter.")
    return []

logger.info(f"Lancement des interprétations SÉQUENTIELLES pour {len(all_visuals)} éléments")

updated_visuals = [] # Nouvelle liste pour stocker les résultats mis à jour

for i, vis in enumerate(all_visuals):
    # Générer un ID unique si nécessaire pour le mapping des résultats
    if 'id' not in vis: # Assigner un ID si manquant
        vis_prefix = os.path.splitext(vis.get("filename", "unknown"))[0] if 'filename' in vis else ''
        vis['id'] = f"{vis_prefix}_{i}_{int(time.time()*1000)}"

    vis_id = vis['id'] # Utiliser l'ID assigné

    # Vérifier que l'image est disponible en base64
    if 'base64' not in vis or not vis['base64']:
        logger.warning(f"({i+1}/{len(all_visuals)}) Image manquante pour {vis_id}. Interprétation impossible: image non disponible")
        updated_visuals.append(vis) # Ajouter l'élément avec l'erreur
        continue # Passe à la visualisation suivante

    logger.info(f"({i+1}/{len(all_visuals)}) Interprétation de : {vis_id}")
    try:
        # Appel DIRECT à la fonction d'interprétation
        interpretation = interpret_visualization_with_gemini(
            vis,
            agent1_data,
            model,
            backend,
            prompts_log_path,
            timeout # Passer le timeout
        )
        vis['interpretation'] = interpretation
        logger.info(f"({i+1}/{len(all_visuals)}) Interprétation reçue pour {vis_id}.")
    except Exception as e:
        logger.error(f"({i+1}/{len(all_visuals)}) Erreur lors de l'interprétation directe: {e}")
        vis['interpretation'] = f"Erreur d'interprétation (directe): {e}"

    updated_visuals.append(vis) # Ajouter l'élément traité (avec succès ou erreur)

logger.info(f"Toutes les {len(all_visuals)} interprétations séquentielles sont terminées")

return updated_visuals

# --- FIN REMPLACEMENT FONCTION ---
def attempt_execution_loop(code: str, csv_file: str, agent1_data: dict, model: str, prompts_log_path: str, backend: str, timeout: int):
    """

```

Tente d'exécuter le code et, en cas d'erreur, demande une correction au LLM en utilisant

Args:

```
code: Code Python à exécuter
csv_file: Chemin absolu du fichier CSV
agent1_data: Données de l'agent1
model: Modèle LLM à utiliser
prompts_log_path: Chemin pour sauvegarder les prompts
backend: Backend pour les appels LLM ('ollama' ou 'gemini')
```

Returns:

Dictionnaire contenant les résultats de l'exécution

"""

```
import base64
from collections import deque
import time
import shutil
import os
import tempfile
import json
import subprocess
import logging
from datetime import datetime
```

```
logger = logging.getLogger("agent2")
```

```
# Initialiser le modèle courant (sera mis à jour si on bascule vers le modèle puissant)
current_model = model
```

```
# Vérifier si le code utilise le bon chemin CSV
```

```
if csv_file not in code:
```

```
    logger.warning(f"Le code initial ne semble pas utiliser le chemin absolu '{csv_file}'")
```

```
# Créer les répertoires temporaires pour l'exécution
```

```
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
```

```
temp_dir = tempfile.mkdtemp(prefix=f"analysis_{timestamp}_")
```

```
# Ajouter un répertoire pour les tables de régression
```

```
tables_dir = os.path.join(temp_dir, "tables")
```

```
os.makedirs(tables_dir, exist_ok=True)
```

```
# Ajouter un répertoire pour les visualisations
```

```
vis_dir = os.path.join(temp_dir, "visualisations")
```

```
os.makedirs(vis_dir, exist_ok=True)
```

```
# Créer un répertoire pour sauvegarder les versions du code
```

```
code_versions_dir = os.path.join("outputs", f"code_versions_{timestamp}")
```

```
os.makedirs(code_versions_dir, exist_ok=True)
```

```
logger.info(f"Les versions de code seront sauvegardées dans {code_versions_dir}")
```

```
import textwrap
```

```
# Code pour capturer les visualisations - MODIFIÉ AVEC SUPPORT CSV AMÉLIORÉ
```

```
vis_code = textwrap.dedent(f"""
```

```
# Ajout pour capturer les visualisations
```

```
import os
```

```
import logging
```

```
import io
```

```
import re
```

```
import matplotlib
```

```
import matplotlib.pyplot as plt
```

```
from matplotlib.figure import Figure
```

```
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
```

```

import json
import numpy as np

# Configuration pour un meilleur rendu - Utilisation d'un style compatible avec les versions
try:
    plt.style.use('whitegrid') # Pour seaborn récent
except:
    try:
        plt.style.use('seaborn') # Fallback vers le style seaborn de base
    except:
        pass # Utiliser le style par défaut si seaborn n'est pas disponible

# Définition des répertoires de sortie
VIS_DIR = r"{vis_dir}"
TABLES_DIR = r"{tables_dir}"
os.makedirs(VIS_DIR, exist_ok=True)
os.makedirs(TABLES_DIR, exist_ok=True)

# Configuration du logger
vis_logger = logging.getLogger("visualisation_capture")

# Redirection de l'affichage pour capturer les sorties
from io import StringIO
import sys

# Classe pour capturer stdout
class CaptureOutput:
    def __init__(self):
        self.old_stdout = sys.stdout
        self.buffer = StringIO()

    def __enter__(self):
        sys.stdout = self.buffer
        return self.buffer

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout = self.old_stdout

# Classe pour convertir les données numpy en JSON
class NumpyEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        if isinstance(obj, np.integer):
            return int(obj)
        if isinstance(obj, np.floating):
            return float(obj)
        return super(NumpyEncoder, self).default(obj)

# Remplacer la fonction plt.show() pour enregistrer les figures
_original_show = plt.show
_fig_counter = 0

def _custom_show(*args, **kwargs):
    global _fig_counter
    try:
        fig = plt.gcf()
        if fig.get_axes():
            _fig_counter += 1
            filepath = os.path.join(VIS_DIR, f"figure_{{_fig_counter}}.png")

            # Capturer les données du graphique actuel
            fig_data = {{{}}}

            # MODIFICATION: Capturer les informations sur les axes et le titre

```

```

fig_title = fig.get_label() or f"Figure {{_fig_counter}}"
fig_data["title"] = fig_title

# MODIFICATION: Essayer de capturer les données utilisées pour la visualisation
for ax_idx, ax in enumerate(fig.get_axes()):
    ax_data = {}

    # Capturer les titres et labels d'axes
    x_label = ax.get_xlabel() or "Axe X"

    y_label = ax.get_ylabel() or "Axe Y"
    ax_title = ax.get_title() or fig_title

    ax_data["x_label"] = x_label
    ax_data["y_label"] = y_label
    ax_data["title"] = ax_title

    # Capturer les données des lignes
    for line_idx, line in enumerate(ax.get_lines()):
        line_id = f"ax{{ax_idx}}_line{{line_idx}}"
        x_data = line.get_xdata()
        y_data = line.get_ydata()

        # Créer un petit dataframe et le convertir en CSV avec contexte
        if len(x_data) > 0 and len(y_data) > 0:
            try:
                import pandas as pd
                # Utiliser les labels des axes comme noms de colonnes
                line_df = pd.DataFrame({{
                    x_label: x_data,
                    y_label: y_data
                }})

                # CORRECTION: Récupérer le titre de l'axe depuis ax_data ou utiliser fig_title
                current_ax_title = ax_data.get("title", fig_title)

                # Ajouter des métadonnées en commentaire CSV
                csv_header = f"# {{current_ax_title}} - Relation entre {{x_label}} et {{y_label}}\n"
                csv_header += f"# Source: Figure {{_fig_counter}}, Ligne {{line_idx}}\n"

                # Ajouter des statistiques en commentaire CSV
                if hasattr(x_data, 'min') and hasattr(y_data, 'min'):
                    stats = f"# Statistiques {{x_label}}: min={{x_data.min():.2f}}\n"
                    stats += f"# Statistiques {{y_label}}: min={{y_data.min():.2f}}\n"
                    csv_header += stats

                csv_data = csv_header + line_df.to_csv(index=False)

                ax_data[line_id] = {{
                    "type": "line",
                    "x": x_data.tolist() if hasattr(x_data, 'tolist') else list(x_data),
                    "y": y_data.tolist() if hasattr(y_data, 'tolist') else list(y_data),
                    "label": line.get_label() if line.get_label() != '_nolegend' else '',
                    "x_label": x_label, # Ajout du label de l'axe x
                    "y_label": y_label, # Ajout du label de l'axe y
                    "title": current_ax_title, # Ajout du titre (corrigé)
                    "csv_data": csv_data
                }}
            except Exception as e:
                vis_logger.error(f"Erreur lors de la conversion des données en CSV: {e}")
                ax_data[line_id] = {{
                    "type": "line",
                    "x": x_data.tolist() if hasattr(x_data, 'tolist') else list(x_data),
                    "y": y_data.tolist() if hasattr(y_data, 'tolist') else list(y_data),
                    "label": line.get_label() if line.get_label() != '_nolegend' else '',
                    "x_label": x_label, # Ajout du label de l'axe x
                    "y_label": y_label, # Ajout du label de l'axe y
                    "title": current_ax_title, # Ajout du titre (corrigé)
                    "csv_data": csv_data
                }}

```

```

        "y": y_data.tolist() if hasattr(y_data, 'tolist') else list(
        "label": line.get_label() if line.get_label() != '_nolegend_'
        "x_label": x_label,
        "y_label": y_label
    }}

# Extraire les lignes, barres, etc.
for line_idx, line in enumerate(ax.get_lines()):
    line_id = f"ax{{{ax_idx}}}_line{{{line_idx}}}"
    if line_id not in ax_data: # Si n'est pas déjà ajouté avec CSV
        ax_data[line_id] = {
            "type": "line",
            "x": line.get_xdata().tolist() if hasattr(line, 'get_xdata') else
            "y": line.get_ydata().tolist() if hasattr(line, 'get_ydata') else
            "label": line.get_label() if line.get_label() != '_nolegend_' else
            "x_label": x_label,
            "y_label": y_label
        }

# Barres
for container_idx, container in enumerate(ax.containers):
    if isinstance(container, matplotlib.container.BarContainer):
        container_id = f"ax{{{ax_idx}}}_bars{{{container_idx}}}"
        rectangles = container.get_children()
        bar_data = {
            "type": "bar",
            "heights": [],
            "positions": [],
            "widths": [],
            "x_label": x_label,
            "y_label": y_label
        }
        for rect in rectangles:
            if hasattr(rect, 'get_height') and hasattr(rect, 'get_x') and ha
                bar_data["heights"].append(rect.get_height())
                bar_data["positions"].append(rect.get_x())
                bar_data["widths"].append(rect.get_width())
        ax_data[container_id] = bar_data

# Essayer de convertir les données de barres en CSV avec contexte
try:
    import pandas as pd

    # Créer un DataFrame pour les barres avec noms personnalisés
    bar_df = pd.DataFrame({
        x_label: bar_data["positions"],
        f"{{{y_label}}}" (hauteur)": bar_data["heights"],

        "Largeur": bar_data["widths"]
    })

# CORRECTION: Récupérer le titre de l'axe depuis ax_data ou utiliser
current_ax_title = ax_data.get("title", f"Figure {{{_fig_counter}}}")

# Ajouter des métadonnées en commentaire CSV
csv_header = f"# {{{current_ax_title}}} - Graphique à barres\\n"
csv_header += f"# Source: Figure {{{_fig_counter}}}, Container {{{c
csv_header += f"# X: {{{x_label}}}, Y: {{{y_label}}}\n"

# Ajouter des statistiques
if len(bar_data["heights"]) > 0:
    heights = np.array(bar_data["heights"])
    stats = f"# Statistiques hauteurs: min={{{heights.min():.2f}}}"

```

```

        csv_header += stats

        ax_data[container_id]["csv_data"] = csv_header + bar_df.to_csv(i)
    except Exception as e:
        vis_logger.error(f"Erreur lors de la conversion des données de b

# Collections (scatter plots, heatmaps)
for collection_idx, collection in enumerate(ax.collections):
    collection_id = f"ax{{{ax_idx}}}_collection{{{collection_idx}}}"
    collection_data = {"type": "collection", "x_label": x_label, "y_label":

# Points (scatter)
if hasattr(collection, 'get_offsets'):
    offsets = collection.get_offsets()
    if len(offsets) > 0:
        collection_data["points"] = offsets.tolist() if hasattr(offsets,

# Convertir en CSV amélioré avec contexte
try:
    import pandas as pd
    offsets_array = np.array(offsets)
    if offsets_array.shape[1] == 2: # Si ce sont des points 2D
        # Utiliser les labels des axes comme noms de colonnes
        points_df = pd.DataFrame({{
            x_label: offsets_array[:, 0],
            y_label: offsets_array[:, 1]
        }})

    # CORRECTION: Récupérer le titre de l'axe depuis ax_data
    current_ax_title = ax_data.get("title", f"Figure {{{_fig_

# Ajouter des métadonnées en commentaire CSV
csv_header = f"# {{{current_ax_title}}} - Nuage de points\
csv_header += f"# Source: Figure {{{_fig_counter}}}, Colle

# Ajouter des statistiques
x_stats = f"# Statistiques {{{x_label}}}: min={{offsets_ar
y_stats = f"# Statistiques {{{y_label}}}: min={{offsets_ar
csv_header += x_stats + y_stats

        collection_data["csv_data"] = csv_header + points_df.to_
    except Exception as e:
        vis_logger.error(f"Erreur lors de la conversion des points e

# Couleurs
if hasattr(collection, 'get_array') and collection.get_array() is not No
    collection_data["values"] = collection.get_array().tolist() if hasat

ax_data[collection_id] = collection_data

# Tentative d'extraire les étiquettes des axes
if ax.get_title():
    ax_data["title"] = ax.get_title()
if ax.get_xlabel():
    ax_data["xlabel"] = ax.get_xlabel()
if ax.get_ylabel():
    ax_data["ylabel"] = ax.get_ylabel()

# Tentative d'extraire les limites des axes
if ax.get_xlim():
    ax_data["xlim"] = ax.get_xlim()
if ax.get_ylim():
    ax_data["ylim"] = ax.get_ylim()

fig_data[f"axes{{{ax_idx}}}"] = ax_data

```

```

# Information globale sur la figure
if fig.get_label():
    fig_data["title"] = fig.get_label()
else:
    fig_data["title"] = f"Figure {{_fig_counter}}"

# Essayer d'extraire directement un dataframe de la figure si possible
fig_csv_data = ""
try:
    # Parcourir les objets pour trouver un DataFrame
    for ax in fig.get_axes():
        for line in ax.get_lines():
            if hasattr(line, '_data_source') and hasattr(line._data_source, 'to_
                title = fig_data.get("title", "Figure")
                x_label = ax.get_xlabel() or "X"
                y_label = ax.get_ylabel() or "Y"

                # Créer un en-tête CSV avec métadonnées
                csv_header = f"# {{title}} - Données source\\n"

                csv_header += f"# Axes: {{x_label}} vs {{y_label}}\\n"
                # Ajouter des statistiques si possible
                if hasattr(line._data_source, 'describe'):
                    try:
                        describe = line._data_source.describe().to_dict()
                        for col, stats in describe.items():
                            csv_header += f"# Statistiques {{col}}: min={{stats.
                    except:
                        pass

                fig_csv_data = csv_header + line._data_source.to_csv(index=False)
                break
except Exception as e:
    vis_logger.error(f"Erreur lors de l'extraction directe du DataFrame: {{e}}")

if fig_csv_data:
    fig_data["csv_data"] = fig_csv_data

try:
    plt.savefig(filepath, bbox_inches='tight', dpi=100)
    vis_logger.info(f"Visualisation sauvegardée : {{filepath}}")

    # Sauvegarde des données dans un fichier JSON
    data_filepath = os.path.join(VIS_DIR, f"figure_{{_fig_counter}}_data.json")
    with open(data_filepath, 'w', encoding='utf-8') as f:
        json.dump(fig_data, f, ensure_ascii=False, indent=2, cls=NumpyEncoder)
    vis_logger.info(f"Données de visualisation sauvegardées : {{data_filepath}}")

    # Sauvegarde des données CSV si disponibles
    if any(["csv_data" in data for ax_data in fig_data.values() if isinstance(ax
        for data in ax_data.values() if isinstance(data, dict)]) or "csv_data"
        csv_filepath = os.path.join(VIS_DIR, f"figure_{{_fig_counter}}_data.csv")

    # Choisir la première source de données CSV disponible
    csv_content = fig_data.get("csv_data", "")
    if not csv_content:
        for ax_data in fig_data.values():
            if isinstance(ax_data, dict):
                for data in ax_data.values():
                    if isinstance(data, dict) and "csv_data" in data:
                        csv_content = data["csv_data"]
                        break
    if csv_content:

```

```

        break

    if csv_content:
        with open(csv_filepath, 'w', encoding='utf-8') as f:
            f.write(csv_content)
        vis_logger.info(f"Données CSV améliorées sauvegardées : {{csv_filepa

except Exception as e:
    vis_logger.error(f"Impossible de sauvegarder la figure {{_fig_counter}} à {{

    # On permet l'affichage pour débogage mais ça sera ignoré dans l'environnement n
    _original_show(*args, **kwargs)
except Exception as e:
    vis_logger.error(f"Erreur dans _custom_show: {{e}}")

plt.show = _custom_show

# Fonction pour capturer les sorties de régression OLS de statsmodels
_original_print = print
def _custom_print(*args, **kwargs):
    output = " ".join(str(arg) for arg in args)
    _original_print(*args, **kwargs) # Affiche normalement

# Détecter si c'est un résultat de régression OLS
if "OLS Regression Results" in output and "=" * 10 in output:
    try:
        # Extraire la table de régression complète
        global _tables_counter
        if '_tables_counter' not in globals():
            _tables_counter = 0
        _tables_counter += 1

        # Sauvegarder le texte
        filepath = os.path.join(TABLES_DIR, f"regression_{{_tables_counter}}.txt")
        with open(filepath, "w") as f:
            f.write(output)

        # Créer une visualisation de la table
        img_path = os.path.join(TABLES_DIR, f"regression_{{_tables_counter}}.png")
        plt.figure(figsize=(12, 10))
        plt.text(0.01, 0.99, output, family='monospace', fontsize=10, verticalalignment='top')
        plt.axis('off')
        plt.tight_layout()
        plt.savefig(img_path, dpi=100, bbox_inches='tight')
        plt.close()

        # Extraire et sauvegarder les données de régression
        # Extraire R-squared
        r_squared_match = re.search(r"R-squared:\\s*([\\d\\.]+)", output)
        r_squared = r_squared_match.group(1) if r_squared_match else "N/A"

        # Extraire les variables et coefficients
        coef_section = re.search(r"==+\\s*\\n\\s*(coef.*?)(?:\\n\\s*==+|\\nZ)", output, re.DOTALL)
        regression_data = {{
            "r_squared": r_squared,
            "coefficients": []
        }}

        if coef_section:
            lines = coef_section.group(1).split('\\n') # Utilisation de \\n car c'est da
            headers = None

            for line in lines:

```

```

        if "coef" in line.lower():
            # C'est la ligne d'en-tête
            headers = re.split(r'\\s{{2,}}', line.strip()) # Utilisation de \\s{
        elif line.strip() and headers:
            # C'est une ligne de données
            values = re.split(r'\\s{{2,}}', line.strip()) # Utilisation de \\s{
            if len(values) >= len(headers):
                variable = values[0]
                coef_data = {
                    "variable": variable,
                    "coef": values[1] if len(values) > 1 else "N/A",
                    "std_err": values[2] if len(values) > 2 else "N/A",
                    "p_value": values[4] if len(values) > 4 else "N/A"
                }
                regression_data["coefficients"].append(coef_data)

# NOUVELLE SECTION: Convertir les coefficients en CSV amélioré
try:
    import pandas as pd
    if regression_data["coefficients"]:
        coef_df = pd.DataFrame(regression_data["coefficients"])

        # Ajouter une colonne de significativité pour faciliter l'interprétation
        try:
            coef_df['significatif'] = coef_df['p_value'].astype(float) < 0.05
        except:
            # Si conversion en float échoue, on continue sans cette colonne
            pass

        # Ajouter des métadonnées contextuelles en commentaire
        csv_header = f"# Résultats de régression OLS - Table {{_tables_counter}}"
        csv_header += f"# R²: {{r_squared}}\\n"
        csv_header += f"# Interprétation: un coefficient positif indique une rel
        csv_header += f"# Une p-value < 0.05 indique que le coefficient est stat

        regression_data["csv_data"] = csv_header + coef_df.to_csv(index=False)

        # Sauvegarder aussi en fichier CSV amélioré
        csv_path = os.path.join(TABLES_DIR, f"regression_{{_tables_counter}}_coe
        with open(csv_path, 'w', encoding='utf-8') as f:
            f.write(regression_data["csv_data"])
        vis_logger.info(f"Données CSV améliorées de coefficients sauvegardées: {
except Exception as e:
    vis_logger.error(f"Erreur lors de la conversion des coefficients en CSV amél

# Sauvegarder les données
data_path = os.path.join(TABLES_DIR, f"regression_{{_tables_counter}}_data.json"
with open(data_path, 'w', encoding='utf-8') as f:
    json.dump(regression_data, f, ensure_ascii=False, indent=2)

vis_logger.info(f"Table de régression sauvegardée : {{filepath}}")
vis_logger.info(f"Données de régression sauvegardées : {{data_path}}")
except Exception as e:
    vis_logger.error(f"Erreur lors de la capture d'une table de régression: {{e}}")

# Remplacer la fonction print
print = _custom_print

# Fonction manuelle pour sauvegarder des figures
def save_figure(fig, name):
    filepath = os.path.join(VIS_DIR, f"{{name}}.png")
    fig.savefig(filepath, bbox_inches='tight', dpi=100)
    vis_logger.info(f"Figure sauvegardée manuellement: {{filepath}}")
    return filepath
""") # Fin de la string vis_code

```

```

# Initialiser les variables pour la boucle d'exécution
temp_filename = os.path.join(temp_dir, "analysis_script.py")
attempt = 0
llm_correction_attempt = 0 # Compteur spécifique pour les tentatives de correction par
execution_results = {"success": False}
execution_output = ""
all_code_versions = []
recent_errors = deque(maxlen=3)
max_attempts = 10
current_stderr = ""

# Flag pour le modèle puissant
tried_powerful_model = False

# Boucle principale d'exécution
while attempt < max_attempts:
    attempt += 1
    logger.info(f"--- Tentative d'exécution {attempt}/{max_attempts} ---")

    # Sanitize le code avant l'exécution
    try:
        valid_columns = agent1_data["metadata"].get("noms_colonnes", [])
        sanitized_code = sanitize_code(code, csv_file, valid_columns)
        if sanitized_code != code:
            logger.info("Code assaini (chemin CSV, colonnes, df_numeric).")
            code = sanitized_code
    except Exception as e:

        logger.error(f"Erreur pendant l'assainissement du code: {e}. Tentative avec le c

# Sauvegarder la version actuelle du code
current_code_path = os.path.join(code_versions_dir, f"attempt_{attempt}.py")
try:
    with open(current_code_path, "w", encoding="utf-8") as f:
        f.write(code)
    all_code_versions.append(current_code_path)
    logger.info(f"Version {attempt} du code sauvegardée: {current_code_path}")
except Exception as e:
    logger.error(f"Impossible de sauvegarder la version {attempt} du code : {e}")

# Par celles-ci:
import textwrap
vis_code_dedented = textwrap.dedent(vis_code) # Conserve l'indentation relative
modified_code = vis_code_dedented + "\n\n" + code.strip()

# Écrire le code modifié dans un fichier temporaire
try:
    with open(temp_filename, "w", encoding="utf-8") as f:
        f.write(modified_code)
except Exception as e:
    logger.error(f"Impossible d'écrire le script temporaire {temp_filename}: {e}")
    execution_results = {"success": False, "error": "Erreur écriture fichier tempora
    break

    # <<< AJOUTER CE BLOC DE DÉBOGAGE >>>
    logger.info(f"--- DEBUG: Vérification du contenu écrit dans {temp_filename} ---")
    try:
        with open(temp_filename, "r", encoding="utf-8") as f_read:
            lines_to_check = 5
            lines = []
            for i in range(lines_to_check):
                try:
                    lines.append(next(f_read).rstrip('\n'))

```

```

        except StopIteration:
            break # Fin du fichier
        logger.info(f"Premières {len(lines)} lignes (espaces représentés par '.', ta
    for i, line in enumerate(lines):
        # Représentation visuelle des espaces/tabs
        line_repr = line.replace(' ', '.').replace('\t', '\\t')
        logger.info(f"  Ligne {i+1}: [{line_repr}]")
except Exception as read_err:
    logger.error(f"  Erreur de lecture du fichier temporaire pour débogage: {read_err}")
logger.info("--- FIN DEBUG ---")
# <<< FIN DU BLOC DE DÉBOGAGE >>>

# Exécuter le code
try:
    logger.info(f"Exécution de: python3 {temp_filename}")
    result = subprocess.run(
        ["python3", temp_filename],
        capture_output=True, text=True, check=True, timeout=300
    )
    logger.info("Exécution réussie")
    logger.debug(f"Sortie standard (stdout): \n{result.stdout}")
    execution_output = result.stdout
    execution_results = {
        "success": True,
        "output": result.stdout,
        "temp_dir": temp_dir,
        "vis_dir": vis_dir,
        "tables_dir": tables_dir,
        "script_path": temp_filename,
        "all_code_versions": all_code_versions,
        "final_code_used_path": current_code_path
    }

# Traiter les visualisations
vis_files = []
if os.path.exists(vis_dir):
    logger.info(f"Recherche de visualisations dans {vis_dir}")
    for file in sorted(os.listdir(vis_dir)):
        if file.lower().endswith(('.png', '.jpg', '.jpeg', '.svg')):
            file_path = os.path.join(vis_dir, file)
            try:
                with open(file_path, 'rb') as img_file:
                    img_data = base64.b64encode(img_file.read()).decode('utf-8')
                # Vérifier la taille de l'image pour le débogage
                file_size = os.path.getsize(file_path)
                logger.info(f"  -> Visualisation trouvée: {file}, taille: {file_size}")

                # Chercher les données associées
                fig_name = os.path.splitext(file)[0]
                data_path = os.path.join(vis_dir, f"{fig_name}_data.json")
                chart_data = None
                if os.path.exists(data_path):
                    try:
                        with open(data_path, 'r', encoding='utf-8') as f:
                            chart_data = json.load(f)
                        logger.info(f"  -> Données associées trouvées pour: {file}")
                    except Exception as e:
                        logger.error(f"Impossible de lire les données associées")

                # Chercher les données CSV associées
                csv_path = os.path.join(vis_dir, f"{fig_name}_data.csv")
                csv_data = ""
                if os.path.exists(csv_path):
                    try:

```

```

        with open(csv_path, 'r', encoding='utf-8') as f:
            csv_data = f.read()
            logger.info(f" -> Données CSV trouvées pour: {file}")
        except Exception as e:
            logger.error(f"Impossible de lire les données CSV associées à {file}")

# Si pas de fichier CSV, essayer d'extraire des données CSV imbriquées
if not csv_data and chart_data:
    # Parcourir chart_data pour trouver csv_data
    for axes_key, axes_value in chart_data.items():
        if isinstance(axes_value, dict):
            for line_key, line_value in axes_value.items():
                if isinstance(line_value, dict) and 'csv_data' in line_value:
                    csv_data = line_value['csv_data']
                    logger.info(f" -> Données CSV intégrées trouvées dans {file}")
                    break
            if csv_data:
                break

vis_files.append({
    'filename': file,
    'path': file_path,
    'base64': img_data,
    'title': chart_data.get('title', ' ').join(os.path.splitext(file)[0]),
    'size': file_size,
    'data': chart_data,
    'csv_data': csv_data # Conserver les données CSV pour comparaison
})
except Exception as e:
    logger.error(f"Impossible de lire ou encoder l'image {file_path}")
else:
    logger.warning(f"Répertoire de visualisations non trouvé: {vis_dir}")

# Capturer les tables de régression
regression_outputs = capture_regression_outputs(result.stdout, tables_dir)

# Vérifier également si des tables ont été sauvegardées directement par le script
if os.path.exists(tables_dir):
    logger.info(f"Recherche de tables de régression dans {tables_dir}")
    for file in sorted(os.listdir(tables_dir)):
        # Ne traiter que les fichiers texte qui ne sont pas déjà traités
        if file.lower().endswith('.txt') and not any(r['text_path'].endswith(file) for r in regression_outputs):
            text_path = os.path.join(tables_dir, file)
            img_path = os.path.join(tables_dir, os.path.splitext(file)[0] + '.png')

            # Lire le contenu du fichier texte
            try:
                with open(text_path, 'r', encoding='utf-8') as f:
                    table_content = f.read()

                # Vérifier si c'est une table de régression OLS
                if "OLS Regression Results" in table_content:
                    table_id = os.path.splitext(file)[0]

                    # Créer l'image si elle n'existe pas
                    if not os.path.exists(img_path):
                        plt.figure(figsize=(12, 10))
                        plt.text(0.01, 0.99, table_content, family='monospace',
                                fontdict=dict(fontsize=10))
                        plt.axis('off')
                        plt.tight_layout()
                        plt.savefig(img_path, dpi=100, bbox_inches='tight')
                        plt.close()

```

```

# Encoder l'image en base64
with open(img_path, 'rb') as img_file:
    img_data = base64.b64encode(img_file.read()).decode('utf-8')

# Extraire des métadonnées basiques
r_squared_match = re.search(r"R-squared:\s*([\d\.]+)", table_text)
r_squared = r_squared_match.group(1) if r_squared_match else None

# Chercher le fichier de données JSON associé
data_path = os.path.join(tables_dir, f"{table_id}_data.json")
regression_data = None
if os.path.exists(data_path):
    try:
        with open(data_path, 'r', encoding='utf-8') as f:
            regression_data = json.load(f)
        logger.info(f" -> Données de régression trouvées pour la table {table_id}")
    except Exception as e:
        logger.error(f"Impossible de lire les données de régression pour la table {table_id}")

# Chercher les données CSV des coefficients
csv_path = os.path.join(tables_dir, f"{table_id}_coefficients.csv")
csv_data = ""
if os.path.exists(csv_path):
    try:
        with open(csv_path, 'r', encoding='utf-8') as f:
            csv_data = f.read()
        logger.info(f" -> Données CSV des coefficients trouvées pour la table {table_id}")
    except Exception as e:
        logger.error(f"Impossible de lire les données CSV pour la table {table_id}")

# Si pas de fichier CSV mais données JSON disponibles
if not csv_data and regression_data and "csv_data" in regression_data:
    csv_data = regression_data["csv_data"]
    logger.info(f" -> Données CSV intégrées trouvées pour la table {table_id}")

# Vérifier la taille de l'image pour le débogage
file_size = os.path.getsize(img_path)

logger.info(f" -> Table de régression trouvée: {table_id}, taille: {file_size} octets")

regression_outputs.append({
    'type': 'regression_table',
    'id': table_id,
    'text_path': text_path,
    'image_path': img_path,
    'base64': img_data,
    'title': f"Résultats de Régression: {table_id.replace('_', ' ')}",
    'metadata': {
        'r_squared': r_squared
    },
    'size': file_size,
    'data': regression_data,
    'csv_data': csv_data # Conserver les données CSV pour comparaison
})
except Exception as e:
    logger.error(f"Erreur lors du traitement de la table {table_id}: {e}")

# Mettre à jour les résultats
execution_results["visualisations"] = vis_files
execution_results["regressions"] = regression_outputs
logger.info(f"{len(vis_files)} visualisation(s) et {len(regression_outputs)} tables de régression")

# Journaliser plus de détails sur les visualisations pour le débogage

```

```

for i, vis in enumerate(vis_files):
    logger.info(f"  Visualisation #{i+1}: {vis.get('filename')}, taille: {vis.get('size', 0)}")
    if 'base64' in vis:
        logger.info(f"    Longueur base64: {len(vis['base64'])} caractères")
    if 'data' in vis and vis['data']:
        logger.info(f"    Données présentes: Oui")
    if 'csv_data' in vis and vis['csv_data']:
        logger.info(f"    Données CSV présentes: {len(vis['csv_data'])} caractères")

for i, reg in enumerate(regression_outputs):
    logger.info(f"  Régression #{i+1}: {reg.get('id')}, taille: {reg.get('size', 0)}")
    if 'base64' in reg:
        logger.info(f"    Longueur base64: {len(reg['base64'])} caractères")
    if 'data' in reg and reg['data']:
        logger.info(f"    Données structurées: Oui")
    if 'csv_data' in reg and reg['csv_data']:
        logger.info(f"    Données CSV présentes: {len(reg['csv_data'])} caractères")

break

# Gestion des erreurs d'exécution
except subprocess.TimeoutExpired:
    logger.error("L'exécution a dépassé le délai de 300 secondes.")
    error_message = "TimeoutExpired: Le script a mis trop de temps à s'exécuter."
    recent_errors.append(error_message.strip())
    current_stderr = error_message

except subprocess.CalledProcessError as e:
    stderr_output = e.stderr.strip()
    logger.error(f"Erreur lors de l'exécution (stderr):\n{stderr_output}")
    recent_errors.append(stderr_output)
    current_stderr = e.stderr

# Sauvegarder l'erreur dans un fichier
error_file = os.path.join(code_versions_dir, f"attempt_{attempt}_error.txt")
try:
    with open(error_file, "w", encoding="utf-8") as f:
        f.write(f"ERREUR RENCONTRÉE LORS DE LA TENTATIVE {attempt}:\n\n")
        f.write(current_stderr)
    logger.info(f"Erreur sauvegardée dans: {error_file}")
except Exception as write_err:
    logger.error(f"Impossible de sauvegarder le fichier d'erreur {error_file}")

# MODIFICATION: Utiliser le modèle puissant après 5 tentatives échouées
if attempt >= 10 and not tried_powerful_model and backend == "gemini":
    logger.warning(f"Après {attempt} tentatives échouées, basculement vers le modèle puissant")
    tried_powerful_model = True
    powerful_model = "gemini-2.5-pro-exp-03-25"
    current_model = powerful_model # Mise à jour du modèle courant

# Créer un prompt spécifique pour résoudre les problèmes de formule OLS
powerful_prompt = f"""\
Voici mon script Python qui génère une erreur. Corrige-le.

```python
{code}
```

Erreur:
```
{current_stderr}
```

Je veux uniquement le code Python corrigé, sans explications. Le script est utilisé pour analyser le
PROBLÈME À RÉSOUDRE:

```

- Il y a une erreur de "unterminated string literal" dans une formule OLS
- Le problème est lié aux noms de pays avec espaces et caractères spéciaux dans la formule
- La solution est d'utiliser des backticks (`) pour encadrer chaque nom de variable avec esp
- Exemple: `Pays\_Afrique du Sud` au lieu de Pays\_Afrique du Sud
- OU simplifier la formule en utilisant une technique d'encodage différente pour les pays

```

"""
# Sauvegarder le prompt pour le modèle puissant
save_prompt_to_file(powerful_prompt, prompts_log_path, "Powerful Model after
logger.info(f"Envoi du prompt de correction au modèle puissant {powerful_mod

try:
    powerful_correction = call_llm(prompt=powerful_prompt, model_name=powerf
    corrected_code = extract_code(powerful_correction)
    if corrected_code and len(corrected_code.strip()) > 0:
        code = corrected_code
        logger.info("Code corrigé par le modèle puissant reçu")
        recent_errors.clear()
        continue
    else:
        logger.warning("Le modèle puissant n'a pas renvoyé de code utilisabl
except Exception as powerful_err:
    logger.error(f"Erreur avec le modèle puissant: {powerful_err}")

# Vérifier si les erreurs se répètent
if len(recent_errors) == 3 and len(set(recent_errors)) == 1:
    logger.warning("Trois erreurs identiques consécutives détectées.")

# Si on n'a pas encore essayé le modèle puissant après 3 erreurs identiques
if not tried_powerful_model and backend == "gemini":
    logger.warning("Basculement vers le modèle Gemini Pro puissant pour corr
    tried_powerful_model = True
    powerful_model = "gemini-2.5-pro-exp-03-25"
    current_model = powerful_model # Mise à jour du modèle courant

# Créer un prompt simplifié
powerful_prompt = f"""Voici mon script Python qui génère une erreur. Cor

```python
{code}
```

Erreur:
```
{current_stderr}
```

Je veux uniquement le code Python corrigé, sans explications. Le script est utilisé pour ana
"""

```

```

# Sauvegarder le prompt pour le modèle puissant
save_prompt_to_file(powerful_prompt, prompts_log_path, "Powerful Model C
logger.info(f"Envoi du prompt de correction au modèle puissant {powerful

try:
    powerful_correction = call_llm(prompt=powerful_prompt, model_name=po
    corrected_code = extract_code(powerful_correction)
    if corrected_code and len(corrected_code.strip()) > 0:
        code = corrected_code
        logger.info("Code corrigé par le modèle puissant reçu")
        recent_errors.clear()
        continue
    else:

```

```

        logger.warning("Le modèle puissant n'a pas renvoyé de code utili
except Exception as powerful_err:
    logger.error(f"Erreur avec le modèle puissant: {powerful_err}")

# Si le modèle puissant a échoué ou a déjà été essayé, passer à la correction
logger.warning("Passage à la correction manuelle.")
manual_edit_path = os.path.join(code_versions_dir, f"manual_fix_for_attempt

if not os.path.exists(manual_edit_path):
    try:
        with open(manual_edit_path, "w", encoding="utf-8") as f:
            f.write(code)
        logger.info(f"Fichier de correction manuelle créé : {manual_edit_pat
    except Exception as create_err:
        logger.error(f"Erreur lors de la création du fichier de correction m
    else:
        try:
            import shutil
            shutil.copyfile(manual_edit_path, manual_edit_path + ".bak")
            logger.info(f"Backup du fichier de correction manuelle créé : {manua
        except Exception as copy_err:
            logger.error(f"Erreur lors de la sauvegarde du fichier {manual_edit_

logger.info(f"Modifiez le fichier {manual_edit_path} si nécessaire.")

import sys

# Afficher le message sur stderr (pas stdout)
sys.stderr.write("■■■ Appuyez sur Entrée pour continuer après modification..
sys.stderr.flush() # Assurez-vous que le message s'affiche immédiatement
try:
    input() # Pas de message dans input() pour éviter qu'il aille dans stdo
except EOFError:
    logger.warning("Pas d'entrée utilisateur détectée, poursuite automatique

try:
    with open(manual_edit_path, "r", encoding="utf-8") as f:
        edited_code = f.read()
except Exception as read_err:
    logger.error(f"Erreur lors de la lecture du fichier {manual_edit_path}:
    edited_code = code

if edited_code != code:
    logger.info(f"Code modifié détecté dans {manual_edit_path}. Mise à jour
    code = edited_code
    recent_errors.clear()
    continue

else:
    logger.info(f"Aucune modification détectée dans {manual_edit_path}. Pass

# Tentative de correction automatique par LLM standard
if attempt < max_attempts:
    llm_correction_attempt += 1
    logger.info(f"Tentative de correction LLM #{llm_correction_attempt}")

# Créer le prompt pour la correction
metadata_str = json.dumps(agent1_data["metadata"], indent=2, ensure_ascii=Fa
recall_prompt = f"""
Le contexte suivant concerne l'analyse d'un fichier CSV :
{metadata_str}

Le chemin absolu du fichier CSV est : {csv_file}

```

Assure-toi d'utiliser ce chemin exact dans `pd.read_csv('{csv_file}')`.

Le code Python ci-dessous, complet avec toutes ses fonctionnalités (gestion des visualisations)

-----  
Code Fautif :

```
```python
{code}
```
```

Erreur Rencontrée : {current\_stderr}

TA MISSION : Corrige uniquement l'erreur indiquée sans modifier la logique globale du code.

RENVOIE UNIQUEMENT le code Python corrigé, encapsulé dans un bloc de code délimité par trois backticks.

Fais bien attention à la nature des variables, numériques, catégorielles, etc.

IMPORTANT: Pour les styles dans matplotlib, utilise 'seaborn-v0\_8-whitegrid' au lieu de 'seaborn-whitegrid'

```
"""
    # Sauvegarder le prompt
    save_prompt_to_file(recall_prompt, prompts_log_path, f"Code Correction Attempt {attempt}")
    logger.info(f"Envoi du prompt de correction au LLM via backend '{backend}' à l'heure {current_time}")
```

```
try:
```

```
    # Obtenir et traiter la correction du LLM
```

```
    new_generated_script = call_llm(prompt=recall_prompt, model_name=current_model_name)
```

```
    extracted_code = extract_code(new_generated_script)
```

```
    clean_code = remove_shell_commands(extracted_code)
```

```
    if not clean_code.strip():
```

```
        logger.warning("Le LLM a renvoyé un code vide après nettoyage. Réessayons.")
```

```
    else:
```

```
        code = clean_code # Mettre à jour le code pour la prochaine tentative
```

```
        logger.info("Code corrigé par le LLM reçu et nettoyé.")
```

```
    time.sleep(1)
```

```
except Exception as llm_call_err:
```

```
    logger.error(f"Erreur lors de l'appel au LLM pour correction: {llm_call_err}")
```

```
    # Si l'appel au LLM standard a échoué et qu'on n'a pas encore essayé le modèle puissant
```

```
    if not tried_powerful_model and backend == "gemini":
```

```
        logger.warning("Tentative avec le modèle Gemini Pro puissant après échec du modèle standard.")
```

```
        tried_powerful_model = True
```

```
        powerful_model = "gemini-2.5-pro-exp-03-25"
```

```
        current_model = powerful_model # Mise à jour du modèle courant
```

```
    # Créer un prompt simplifié
```

```
    powerful_prompt = f"""Voici mon script Python qui génère une erreur.
```

```
```python
```

```
{code}
```

```
```
```

Erreur:

```
```
```

```
{current_stderr}
```

```
```
```

Je veux uniquement le code Python corrigé, sans explications. Le script est utilisé pour analyser des données.

Assure-toi d'utiliser 'seaborn-v0\_8-whitegrid' au lieu de 'seaborn-whitegrid' qui est obsolète.

```
"""
```

```
    # Sauvegarder le prompt pour le modèle puissant
```

```
    save_prompt_to_file(powerful_prompt, prompts_log_path, "Powerful Model Prompt")
```

```

        logger.info(f"Envoi du prompt de correction au modèle puissant {powerful_model_name}")

        try:
            powerful_correction = call_llm(prompt=powerful_prompt, model_name=powerful_model_name)
            corrected_code = extract_code(powerful_correction)
            if corrected_code and len(corrected_code.strip()) > 0:
                code = corrected_code
                logger.info("Code corrigé par le modèle puissant reçu")
                recent_errors.clear()
                continue
            else:
                logger.warning("Le modèle puissant n'a pas renvoyé de code u")
        except Exception as powerful_err:
            logger.error(f"Erreur avec le modèle puissant: {powerful_err}")

    # Si aucun LLM ne fonctionne, passer à la correction manuelle en dernier
    logger.warning("Toutes les tentatives de correction automatique ont échoué")
    manual_edit_path = os.path.join(code_versions_dir, f"manual_fix_final_at_{attempt}.txt")

    try:
        with open(manual_edit_path, "w", encoding="utf-8") as f:

            f.write(code)
            logger.info(f"Fichier de correction manuelle finale créé : {manual_edit_path}")

            # Attendre la correction manuelle
            sys.stderr.write("■■■ CORRECTION FINALE MANUELLE REQUISE. Modifiez le code ci-dessous\n")
            sys.stderr.flush()
            try:
                input()
            except EOFError:
                logger.warning("Pas d'entrée utilisateur détectée, poursuite automatique")

            with open(manual_edit_path, "r", encoding="utf-8") as f:
                edited_code = f.read()
            code = edited_code
            recent_errors.clear()

    except Exception as manual_err:
        logger.error(f"Erreur lors de la correction manuelle finale: {manual_err}")
        execution_results = {
            "success": False,
            "error": f"Échec complet - correction automatique et manuelle: {manual_err}",
            "stderr": current_stderr,
            "all_code_versions": all_code_versions
        }
        break

except Exception as general_err:
    logger.error(f"Erreur générale inattendue lors de la tentative {attempt}: {general_err}")
    error_message = f"Erreur générale inattendue: {general_err}"
    recent_errors.append(error_message.strip())
    execution_results = {
        "success": False,
        "error": error_message,
        "stderr": current_stderr if current_stderr else str(general_err),
        "all_code_versions": all_code_versions
    }
    break

# Ajouter les interprétations pour les visualisations et tables de régression
if execution_results.get("success"):
    all_visuals = []

```

```

# Traiter les visualisations
if "visualisations" in execution_results:
    all_visuals.extend(execution_results["visualisations"])

# Traiter les tables de régression
if "regressions" in execution_results:
    all_visuals.extend(execution_results["regressions"])

if all_visuals:
    logger.info("Génération d'interprétations pour les visualisations et tables avec")
    all_visuals_with_interpretations = generate_visualization_interpretations(
        execution_results.get("visualisations", []),
        execution_results.get("regressions", []),
        agent1_data,
        current_model, # Utilisation du modèle courant (qui peut être le modèle précédent)
        backend,
        prompts_log_path
    )

    # Mettre à jour les résultats avec les interprétations
    execution_results["all_visuals"] = all_visuals_with_interpretations

# Finaliser et journaliser les résultats
final_status = "success" if execution_results.get("success") else "failed"
logger.info(f"Fin de la boucle d'exécution. Statut final: {final_status}. Total tentatives: {len(execution_results.get('attempts', []))}")

# Sauvegarder une copie du code final
final_script_path = None
last_code_version_path = all_code_versions[-1] if all_code_versions else None

if last_code_version_path and os.path.exists(last_code_version_path):
    final_script_name = f"analysis_script_{timestamp}_{final_status}.py"
    outputs_dir = "outputs"
    os.makedirs(outputs_dir, exist_ok=True)
    final_script_path = os.path.join(outputs_dir, final_script_name)

    try:
        shutil.copyfile(last_code_version_path, final_script_path)
        logger.info(f"Version finale du code ({final_status}) copiée vers: {final_script_path}")
    except Exception as copy_err:
        logger.error(f"Impossible de copier le script final {last_code_version_path} vers {final_script_path}")
        final_script_path = None
elif not all_code_versions:
    logger.warning("Aucune version de code n'a été générée ou sauvegardée.")
else:
    logger.warning(f"Le dernier fichier de code {last_code_version_path} n'existe pas. I")

# Créer un fichier d'index pour documenter la session
index_file_path = os.path.join(code_versions_dir, "index.txt")
try:
    with open(index_file_path, "w", encoding="utf-8") as f:
        f.write(f"SESSION D'ANALYSE DU {timestamp}\n")
        f.write(f"Fichier CSV: {csv_file}\n")
        f.write(f"Modèle LLM: {model}\n")
        f.write(f"Modèle LLM courant à la fin: {current_model}\n")
        f.write(f"Nombre de versions de code générées/sauvegardées: {len(all_code_versions)}\n")
        f.write(f"Nombre total de tentatives d'exécution effectuées: {attempt}\n")
        f.write(f"Nombre de tentatives de correction LLM: {llm_correction_attempt}\n")

        f.write(f"Modèle puissant utilisé: {'Oui' if tried_powerful_model else 'Non'}\n")
        f.write(f"Résultat final: {'Succès' if execution_results.get('success') else 'Échec'}\n")

    if not execution_results.get("success"):

```

```

        f.write(f"Dernière erreur enregistrée: {execution_results.get('error', 'N/A')}
        last_stderr = execution_results.get('stderr')
        if last_stderr:
            f.write(f"Dernier stderr:\n```\n{last_stderr}\n```\n")

    f.write("\nLISTE DES VERSIONS DE CODE GÉNÉRÉES/UTILISÉES:\n")
    for i, path in enumerate(all_code_versions, 1):
        f.write(f"- Version {i} (Tentative {i}): {os.path.basename(path)}\n")
        error_path = path.replace(".py", "_error.txt")
        if os.path.exists(error_path):
            f.write(f"  - Erreur associée: {os.path.basename(error_path)}\n")
            manual_edit_trigger_path = os.path.join(os.path.dirname(path), f"manual_fix_{i}.txt")
            if os.path.exists(manual_edit_trigger_path):
                f.write(f"    - Fichier pour édition manuelle (créé après échec tentative {i})\n")

    if final_script_path and os.path.exists(final_script_path):
        f.write(f"\nScript final utilisé ({final_status}): {os.path.basename(final_script_path)}\n")
        f.write(f"Chemin complet: {final_script_path}\n")
    else:
        if last_code_version_path and not os.path.exists(last_code_version_path):
            f.write(f"\nScript final non sauvegardé (fichier source {os.path.basename(last_code_version_path)})\n")
        elif final_script_path is None and last_code_version_path:
            f.write(f"\nScript final non sauvegardé (échec de la copie depuis {os.path.basename(last_code_version_path)})\n")
        else:
            f.write(f"\nScript final non sauvegardé (aucune version de code disponible)\n")

    if os.path.exists(prompts_log_path):
        f.write(f"\nJournal des prompts envoyés au LLM: {os.path.basename(prompts_log_path)}\n")
        f.write(f"Chemin complet: {prompts_log_path}\n")

    f.write(f"\nRépertoire des versions de code: {code_versions_dir}\n")
    if execution_results.get("success"):
        vis_dir_final = execution_results.get('vis_dir', os.path.join(temp_dir, "visualisations"))
        tables_dir_final = execution_results.get('tables_dir', os.path.join(temp_dir, "tables"))
        f.write(f"Répertoire des visualisations: {vis_dir_final}\n")
        f.write(f"Répertoire des tables de régression: {tables_dir_final}\n")
        temp_dir_final = execution_results.get('temp_dir', temp_dir)
        f.write(f"Répertoire temporaire d'exécution: {temp_dir_final}\n")
        logger.info(f"Fichier d'index créé: {index_file_path}")
    except Exception as index_err:
        logger.error(f"Impossible de créer le fichier d'index {index_file_path}: {index_err}")
        index_file_path = None

# Ajouter les chemins finaux aux résultats retournés
execution_results["final_script_path"] = final_script_path
execution_results["code_versions_dir"] = code_versions_dir
execution_results["index_file"] = index_file_path
execution_results["current_model"] = current_model # Ajouter le modèle utilisé pour les régressions

# Traiter les tables de régression avec une interprétation dédiée
regression_outputs = execution_results.get("regressions", [])
if regression_outputs:
    logger.info(f"Génération d'interprétations détaillées pour {len(regression_outputs)} régressions")

    for reg in regression_outputs:
        if 'data' in reg:
            # Récupérer le code de régression
            regression_code = reg.get('regression_code', '')

            # Appel à notre nouvelle fonction pour interpréter la régression
            detailed_interpretation = interpret_regression_with_llm(
                reg['data'],
                regression_code,
                agent1_data,
                current_model, # Utiliser le modèle courant qui peut être le modèle précédent
            )

```

```

        backend,
        prompts_log_path,
        timeout=180 # Plus de temps pour l'analyse détaillée
    )

    # Ajouter l'interprétation détaillée au résultat
    reg['detailed_interpretation'] = detailed_interpretation
    logger.info(f"Interprétation détaillée générée pour la régression {reg.get('id')}")
else:
    logger.warning(f"Pas de données structurées pour la régression {reg.get('id')}")

# Mettre à jour les résultats avec les régressions interprétées
execution_results["regressions"] = regression_outputs

return execution_results

# Nouvelle fonction à ajouter dans agent2.py

def interpret_regression_with_llm(regression_data, code_executed, agent1_data, model, backend, prompts_log_path, timeout):
    """
    Interprète de manière détaillée les résultats d'une régression économétrique en utilisant un LLM.

    Args:
        regression_data: Données structurées de la régression (coefficients, r-squared, etc.)
        code_executed: Code Python qui a généré cette régression
        agent1_data: Données de l'agent1 pour le contexte
        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM
        prompts_log_path: Chemin pour sauvegarder les prompts
        timeout: Timeout en secondes. Défaut: 120.

    Returns:
        str: Interprétation détaillée des résultats de régression
    """
    from llm_utils import call_llm
    from datetime import datetime

    # Extraire les informations clés de la régression
    r_squared = regression_data.get('r_squared', 'N/A')
    coefficients = regression_data.get('coefficients', [])

    # Préparer un résumé des coefficients pour le prompt
    coef_summary = ""
    for coef in coefficients:
        var_name = coef.get('variable', 'Inconnu')
        coef_value = coef.get('coef', 'N/A')
        p_value = coef.get('p_value', 'N/A')
        std_err = coef.get('std_err', 'N/A')
        coef_summary += f"{var_name}: coefficient={coef_value}, p-value={p_value}, std_err={std_err}\n"

    # Extraire le contexte de recherche de l'agent1
    user_prompt = agent1_data.get('user_prompt', 'Non disponible')
    introduction = agent1_data.get('llm_output', {}).get('introduction', 'Non disponible')
    hypotheses = agent1_data.get('llm_output', {}).get('hypotheses', 'Non disponible')

    # Extraire les noms des colonnes pour avoir une idée des variables disponibles
    column_names = agent1_data.get('metadata', {}).get('noms_colonnes', [])

    # Créer le prompt pour l'interprétation de la régression
    prompt = f"""## INTERPRÉTATION ÉCONOMÉTRIQUE DÉTAILLÉE

### Résultats de régression
R-squared: {r_squared}

```

```

### Coefficients
{coef_summary}

### Code Python ayant généré cette régression
```python
{code_executed}
```

### Question de recherche
{user_prompt}

### Contexte académique
{introduction[:500]}...

### Hypothèses de recherche
{hypotheses}

### Variables disponibles dans le dataset
{' , '.join(column_names)}

---

En tant qu'économetre expert, ton objectif est de fournir une interprétation précise, rigoureuse et complète de la régression présentée ci-dessus.

Ton interprétation doit inclure:

1. Analyse du modèle global
    - Qualité globale de l'ajustement ( $R^2$ )
    - Validité statistique du modèle
    - Adéquation du modèle à la question de recherche

2. Interprétation des coefficients significatifs
    - Analyse détaillée de chaque coefficient statistiquement significatif ( $p < 0.05$ )
    - Interprétation précise de l'effet marginal (magnitude et direction)
    - Unités de mesure et contexte économique de chaque effet

3. Implications économiques
    - Mécanismes économiques sous-jacents expliquant les relations observées
    - Liens avec les théories économiques pertinentes
    - Implications pour la question de recherche initiale

4. Limites de l'estimation
    - Problèmes potentiels d'endogénéité, de variables omises ou de causalité
    - Robustesse des résultats
    - Pistes d'amélioration du modèle

IMPORTANT:
- Base ton analyse uniquement sur les résultats statistiques fournis, tout en les contextualisant.
- Utilise un langage économétrique précis (élasticités, effets marginaux, significativité, etc.).
- Procède coefficient par coefficient pour les variables significatives.
- Fais le lien entre les résultats statistiques et les mécanismes économiques.
- Ton analyse doit être concise mais complète, en 3-4 paragraphes.

Il est essentiel que cette interprétation soit suffisamment détaillée pour former le cœur d'un article académique.

"""

# Sauvegarder le prompt dans le fichier journal
try:
    with open(prompts_log_path, "a", encoding="utf-8") as f:
        f.write("\n\n" + "="*80 + "\n")
        f.write(f"PROMPT POUR INTERPRÉTATION DE RÉGRESSION - {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
        f.write("\n\n" + "="*80 + "\n")
        f.write(prompt)
        f.write("\n\n" + "="*80 + "\n")
except Exception as e:

```

```

        logger.error(f"Erreur lors de la sauvegarde du prompt: {e}")

    try:
        logger.info(f"Appel au LLM pour l'interprétation détaillée de la régression (R²={r_s}")
        interpretation = call_llm(
            prompt=prompt,
            model_name=model,
            backend=backend,
            timeout=timeout
        )

        logger.info("Interprétation détaillée de la régression générée avec succès")
        return interpretation
    except Exception as e:
        logger.error(f"Erreur lors de l'interprétation détaillée de la régression: {e}")
        return f"Erreur lors de l'interprétation détaillée de la régression: {e}"

# Fonction pour extraire le code qui a généré une régression
def extract_regression_code(full_code, table_content):
    """Tente d'identifier le code qui a généré une régression spécifique."""
    if not full_code:
        return ""

    # Rechercher des patterns communs dans les codes de régression
    import re

    # Extraire les noms de variables potentiels de la table de régression
    var_names = re.findall(r"([A-Za-z0-9_]+\s+[\d\.-]+\s+[\d\.-]+\s+[\d\.-]+\s+[\d\.-]+", table_content)

    # Chercher les modèles OLS qui utilisent ces variables
    regression_patterns = [
        r"(.?sm\.OLS.*?\fit\(\).*?)(?=\n\n|\Z)", # Pattern pour statsmodels OLS
        r"(.?statsmodels\.api.*?OLS.*?\fit\(\).*?)(?=\n\n|\Z)", # Autre pattern pour statsmodels OLS
        r"(.?LinearRegression\(\).*?\fit\(..*?\).*?)(?=\n\n|\Z)" # Pattern pour sklearn LinearRegression
    ]

    # Si nous avons des noms de variables, chercher des lignes qui les utilisent
    if var_names:
        var_patterns = []
        for var in var_names[:5]: # Limiter aux 5 premières variables pour éviter de surcharger
            var_patterns.append(r".*?" + re.escape(var) + r".*?\n")

        # Chercher des blocs de code qui contiennent à la fois un modèle de régression
        # et les variables identifiées
        for reg_pattern in regression_patterns:
            reg_blocks = re.findall(reg_pattern, full_code, re.DOTALL)
            for block in reg_blocks:
                # Vérifier si ce bloc contient au moins une des variables
                for var_pattern in var_patterns:
                    if re.search(var_pattern, block, re.MULTILINE):
                        # Bloc de code qui contient à la fois un modèle de régression et une variable
                        return block.strip()

    # Si nous n'avons pas trouvé de code spécifique, chercher simplement un modèle OLS
    for pattern in regression_patterns:
        matches = re.findall(pattern, full_code, re.DOTALL)
        if matches:
            # Prendre le premier bloc de code qui correspond à un modèle de régression
            return matches[0].strip()

    # Si nous n'avons toujours rien trouvé, retourner une portion du code global
    # qui contient des mots-clés de régression
    regression_keywords = ["OLS", "regression", "statsmodels", "LinearRegression", "fit("]

```

```

lines = full_code.split('\n')
for i, line in enumerate(lines):
    for keyword in regression_keywords:
        if keyword in line:
            # Retourner un bloc de 10 lignes autour de cette ligne si possible
            start = max(0, i - 5)
            end = min(len(lines), i + 6)
            return '\n'.join(lines[start:end])

# Si tout échoue, retourner une chaîne vide
return ""

def generate_analysis_code(csv_file, user_prompt, agent1_data, model, prompts_log_path, backend):
    """
    Génère le code d'analyse économétrique de niveau accessible

    Args:
        csv_file: Chemin absolu du fichier CSV
        user_prompt: Prompt initial de l'utilisateur
        agent1_data: Données de l'agent1
        model: Modèle LLM à utiliser
        prompts_log_path: Chemin pour sauvegarder les prompts
        backend: Backend pour les appels LLM

    Returns:
        str: Code d'analyse généré
    """
    # Préparation du prompt pour le LLM avec le chemin absolu du fichier et les noms de colonnes
    metadata_str = json.dumps(agent1_data["metadata"], indent=2, ensure_ascii=False)
    colonnes = agent1_data["metadata"].get("noms_colonnes", [])
    col_dict = ",\n".join([f"        \"{col}\"": \"{col}\"" for col in colonnes])

    # Extraire les sections si disponibles
    introduction = agent1_data['llm_output'].get('introduction', 'Non disponible')
    literature_review = agent1_data['llm_output'].get('literature_review', 'Non disponible')
    hypotheses = agent1_data['llm_output'].get('hypotheses', 'Non disponible')
    methodology = agent1_data['llm_output'].get('methodology', 'Non disponible')
    limitations = agent1_data['llm_output'].get('limitations', 'Non disponible')
    variables_info = agent1_data['llm_output'].get('variables', 'Non disponible')

    # Utiliser les anciennes sections si les nouvelles ne sont pas disponibles
    if introduction == 'Non disponible':
        introduction = agent1_data['llm_output'].get('problematisation', 'Non disponible')

    if methodology == 'Non disponible':
        methodology = agent1_data['llm_output'].get('approches_suggerees', 'Non disponible')

    if limitations == 'Non disponible':
        limitations = agent1_data['llm_output'].get('points_vigilance', 'Non disponible')

    # Créer le prompt pour la génération de code
    prompt = f"""### GÉNÉRATION DE CODE D'ANALYSE DE DONNÉES

### Fichier CSV et Métadonnées
```json
{metadata_str}
```

### Chemin absolu du fichier CSV
{csv_file}

### Noms exacts des colonnes à utiliser
{colonnes}

```

```
### Introduction et problématique de recherche
{introduction}
```

```
### Hypothèses de recherche
{hypotheses}
```

```
### Méthodologie proposée
{methodology}
```

```
### Limites identifiées
{limitations}
```

```
### Informations sur les variables
{variables_info}
```

```
### Demande initiale de l'utilisateur
{user_prompt}
```

```
---
```

Tu es un analyste de données expérimenté. Ta mission est de générer un script Python d'analyse.

DIRECTIVES:

1. CHARGEMENT ET PRÉTRAITEMENT DES DONNÉES

- Utilise strictement le chemin absolu '{csv\_file}'
- Nettoie les données (valeurs manquantes, outliers)
- Crée des statistiques descriptives claires

2. VISUALISATIONS ATTRAYANTES ET INFORMATIVES

- Crée au moins 4-5 visualisations avec matplotlib/seaborn:
  - \* Matrice de corrélation colorée et lisible
  - \* Distributions des variables principales
  - \* Relations entre variables importantes
  - \* Graphiques adaptés au type de données
- Utilise des couleurs attrayantes et des styles modernes
- Ajoute des titres clairs, des légendes informatives et des ÉTIQUETTES D'AXES EXPLICITES
- IMPORTANT: Assure-toi d'utiliser ax.set\_xlabel() et ax.set\_ylabel() avec des descriptions
- IMPORTANT: Assure-toi que les graphiques soient sauvegardés ET affichés
- Utilise plt.savefig() AVANT plt.show() pour chaque graphique
- IMPORTANT: Pour les styles Seaborn, utilise 'seaborn-v0\_8-whitegrid' au lieu de 'seaborn'

3. MODÉLISATION SIMPLE ET CLAIRE

- Implémente les modèles de régression appropriés
- Utilise statsmodels avec des résultats complets
- Présente les résultats de manière lisible
- Documente clairement chaque étape

4. TESTS DE BASE

- Vérifie la qualité du modèle avec des tests simples
- Analyse les résidus
- Vérifie la multicollinéarité si pertinent

5. CAPTURE ET STOCKAGE DES DONNÉES POUR INTERPRÉTATION

- IMPORTANT: Pour chaque visualisation, stocke le DataFrame utilisé dans une variable
- IMPORTANT: Après chaque création de figure, stocke les données utilisées pour permettre la régénération
- Assure-toi que chaque figure peut être associée aux données qui ont servi à la générer

EXIGENCES TECHNIQUES:

- Utilise pandas, numpy, matplotlib, seaborn, et statsmodels
- Organise ton code en sections clairement commentées
- Utilise ce dictionnaire pour accéder aux colonnes:

```
```python
col = {{
{col_dict}
```

```
}}  
\\,
```

- Document chaque étape de façon simple et accessible
- Pour chaque visualisation:
  - \* UTILISE des titres clairs pour les graphiques et les axes
  - \* SAUVEGARDE avec plt.savefig() PUIS
  - \* AFFICHE avec plt.show()
- Pour les tableaux de régression, utilise print(results.summary())

IMPORTANT:

- Adapte l'analyse aux données disponibles
- Mets l'accent sur les visualisations attrayantes et bien étiquetées
- Assure-toi que chaque graphique a des étiquettes d'axe claires via ax.set\_xlabel() et ax.s
- Assure-toi que chaque graphique est à la fois SAUVEGARDÉ et AFFICHÉ
- Utilise plt.savefig() AVANT plt.show() pour chaque graphique
- IMPORTANT: Pour les styles Seaborn, utilise 'whitegrid' au lieu de 'seaborn-whitegrid' ou  
"""

```
# Sauvegarder le prompt dans le fichier journal
save_prompt_to_file(prompt, prompts_log_path, "Initial Code Generation")
logger.info("Appel LLM pour génération du code d'analyse")
try:
    # Utilisation de llm_utils.call_llm
    generated_output = call_llm(prompt=prompt, model_name=model, backend=backend)

    # Extraire et nettoyer le code généré
    generated_code = extract_code(generated_output)
    clean_code = remove_shell_commands(generated_code)

    # S'assurer que print(results.summary()) est présent
    if "results.summary()" in clean_code and "print(results.summary())" not in clean_code:
        clean_code = clean_code.replace("results.summary()", "print(results.summary())")

    # Assurer que les styles Seaborn sont compatibles
    clean_code = clean_code.replace("seaborn-v0_8-whitegrid", "whitegrid")
    clean_code = clean_code.replace("seaborn-whitegrid", "whitegrid")
    clean_code = clean_code.replace("seaborn-v0_8-white", "white")
    clean_code = clean_code.replace("seaborn-white", "white")
    clean_code = clean_code.replace("seaborn-v0_8-darkgrid", "darkgrid")
    clean_code = clean_code.replace("seaborn-darkgrid", "darkgrid")

    return clean_code

except Exception as e:
    logger.error(f"Erreur lors de l'appel LLM pour génération: {e}")
    sys.exit(1)

def generate_analysis_narrative(execution_results, agent1_data, model):
    """
    Fonction de placeholder pour la génération de narration explicative.
    Cette fonction peut être implémentée pour générer une narration basée sur les résultats.

    Args:
        execution_results: Résultats de l'exécution du code
        agent1_data: Données de l'agent1
        model: Modèle LLM à utiliser

    Returns:
        Dictionnaire avec une narration par défaut
    """
    # Pour l'instant, retourne juste un dictionnaire avec une narration par défaut
    return {
```

```

        "narrative": "Analyse exécutée avec succès. Veuillez consulter les visualisations et
    }

def main():
    """
    Fonction principale qui exécute le pipeline d'analyse économétrique.
    """
    parser = argparse.ArgumentParser(
        description="Agent 2: Analyse économétrique"
    )
    parser.add_argument("csv_file", help="Chemin vers le fichier CSV")
    parser.add_argument("user_prompt", help="Prompt initial de l'utilisateur")
    parser.add_argument("agent1_output", help="Chemin vers le fichier de sortie de l'agent 1")
    parser.add_argument("--model", default="gemini-2.0-flash", help="Modèle LLM à utiliser")
    parser.add_argument("--backend", default="gemini", choices=["ollama", "gemini"], help="Backend")
    parser.add_argument("--auto-confirm", action="store_true", help="Ignore la pause manuelle")
    parser.add_argument("--log-file", help="Fichier de log spécifique") # Nouvel argument
    args = parser.parse_args()

    # Reconfigurer le logging si un fichier de log spécifique est fourni
    if args.log_file:
        # Supprimer les handlers existants
        for handler in logger.handlers[:]:
            logger.removeHandler(handler)

        # Ajouter les nouveaux handlers
        file_handler = logging.FileHandler(args.log_file, mode='a') # mode 'a' pour append
        file_handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s'))
        logger.addHandler(file_handler)

        stream_handler = logging.StreamHandler(sys.stderr)
        stream_handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s'))
        logger.addHandler(stream_handler)

        logger.info(f"Logging redirigé vers {args.log_file}")

    # Créer les répertoires de sortie et les noms de fichiers
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    os.makedirs("outputs", exist_ok=True)
    code_versions_dir = os.path.join("outputs", f"code_versions_{timestamp}")
    prompts_log_path = os.path.join(code_versions_dir, "prompts.txt")

    # Charger les données de l'agent 1
    try:
        with open(args.agent1_output, "r", encoding="utf-8") as f:
            agent1_data = json.load(f)
    except FileNotFoundError:
        logger.error(f"Erreur: Le fichier de l'agent 1 '{args.agent1_output}' n'a pas été trouvé")
        sys.exit(1)
    except json.JSONDecodeError:
        logger.error(f"Erreur: Impossible de décoder le JSON du fichier de l'agent 1 '{args.agent1_output}'")
        sys.exit(1)

    # Générer le code d'analyse économétrique
    logger.info("Génération du code d'analyse économétrique")
    analysis_code = generate_analysis_code(
        args.csv_file,
        args.user_prompt,
        agent1_data,
        args.model,
        prompts_log_path,
        args.backend
    )

```

```

)
if not analysis_code:
    logger.error("La génération du code initial a échoué. Arrêt.")
    sys.exit(1)

# Exécuter le code d'analyse
logger.info("Exécution du code d'analyse")
execution_results = attempt_execution_loop(
    analysis_code,
    args.csv_file,
    agent1_data,
    args.model,
    prompts_log_path,
    args.backend
)

# Générer la narration explicative des résultats
narrative = {"narrative": "L'analyse n'a pas été exécutée avec succès ou a échoué."}
if execution_results.get("success"):
    logger.info("Génération de la narration explicative des résultats")
    narrative = generate_analysis_narrative(
        execution_results,
        agent1_data,
        execution_results.get("current_model", args.model) # Utiliser le modèle courant
    )
else:
    error_msg = execution_results.get('error', 'Erreur inconnue lors de l\'exécution')
    stderr_msg = execution_results.get('stderr')
    full_error = f"L'analyse a échoué: {error_msg}"
    if stderr_msg:
        full_error += f"\nDernier Stderr:\n{stderr_msg[:500]}..."
    narrative = {
        "narrative": full_error
    }
    logger.warning(f"L'exécution du code a échoué. Raison: {error_msg}")

# Combiner toutes les visualisations et tables de régression en une seule liste
all_visuals = execution_results.get("all_visuals", [])
if not all_visuals:
    all_visuals = execution_results.get("visualisations", []) + execution_results.get("r", [])

# Préparer la sortie finale
output = {
    "initial_generated_code": analysis_code,
    "execution_results": execution_results,
    "narrative": narrative.get("narrative"),
    "final_script_path": execution_results.get("final_script_path", "Non disponible"),
    "visualisations": all_visuals, # Liste combinée
    "prompts_log_file": prompts_log_path if os.path.exists(prompts_log_path) else "Non g",
    "output_directory": code_versions_dir,
    "index_file": execution_results.get("index_file", "Non généré ou erreur"),
    "current_model": execution_results.get("current_model", args.model) # Ajouter le mo
}

# Afficher la sortie au format JSON
try:
    print(json.dumps(output, ensure_ascii=False, indent=2))
except TypeError as e:
    logger.error(f"Erreur lors de la sérialisation en JSON: {e}")
    simplified_output = {k: str(v) for k, v in output.items()}
    print(json.dumps(simplified_output, ensure_ascii=False, indent=2))

return 0 if execution_results.get("success") else 1

if __name__ == "__main__":

```

```
sys.exit(main())
```

## 7.4 Code Source: agent3.py

Description:

Agent 3: Synthèse et Rapport PDF Ce script génère un rapport structuré au format PDF à partir des analyses réalisées par les agents 1 et 2. Il assure une progression logique des idées, génère les sections complémentaires et met en forme le rapport final. Usage: `python agent3.py agent1_output agent2_output user_prompt [--model modele] [--backend backend]` Arguments: agent1\_output: Fichier JSON généré par l'agent 1 agent2\_output: Fichier JSON généré par l'agent 2 user\_prompt: Prompt utilisateur original --model: Modèle LLM à utiliser (défaut: gemma3:27b) --backend: Backend LLM ('ollama' ou 'gemini', défaut: 'ollama')

Structure du fichier:

Classes:

- `= "significant" if is_significant else ""` `significance_text = "Significatif" if is_significant else "Non significatif"` `html += f""` except `ValueError`: `html += ""` `significance_text = "Indéterminé"` else: Pas de description

Fonctions:

- `basename_filter`: Filtre Jinja2 pour obtenir le nom de base d'un chemin
- `preprocess_markdown`: Prétraite le texte Markdown pour assurer une meilleure compatibilité avec les convertisseurs
- `markdown_filter`: Filtre Jinja2 amélioré pour convertir du Markdown en HTML avec support avancé
- `csv_to_html_table`: Convertit les données CSV en tableau HTML
- `parse_csv_for_summary`: Extrait des informations résumées des données CSV
- `save_images`: Sauvegarde les visualisations à partir de base64 et retourne une liste de dictionnaires
- `format_regression_results`: Formate les résultats de régression pour une meilleure présentation
- `parse_regression_to_html`: Tente de trouver une table OLS dans le texte et la convertit en HTML
- `generate_comprehensive_economic_analysis`: Génère un raisonnement économique complet en agrégeant toutes les interprétations des visualisations, tables OLS et la problématique initiale
- `generate_comprehensive_visual_analysis`: Génère une analyse globale de toutes les visualisations
- `generate_synthesis`: Génère une synthèse de type résumé simple
- `generate_discussion_section`: Génère une section discussion simple et accessible
- `generate_conclusion_section`: Génère une conclusion simple
- `generate_references`: Génère des références bibliographiques simples
- `interpret_visualization_with_gemini`: Interprète une visualisation en demandant une interprétation concise contextualisée avec la prompt utilisateur
- `generate_report_docx`: Génère un rapport Word au format simple avec python-docx

- `markdown_to_plain_text`: Convertit le Markdown en texte brut
- `generate_report_pdf`: Génère un rapport PDF au format amélioré avec WeasyPrint et Jinja2
- `main`: Fonction principale qui orchestre la génération du rapport

#### Code source complet:

```
#!/usr/bin/env python3
"""
Agent 3: Synthèse et Rapport PDF

Ce script génère un rapport structuré au format PDF à partir des analyses
réalisées par les agents 1 et 2. Il assure une progression logique des idées,
génère les sections complémentaires et met en forme le rapport final.

Usage:
    python agent3.py agent1_output agent2_output user_prompt [--model modele] [--backend backend]

Arguments:
    agent1_output: Fichier JSON généré par l'agent 1
    agent2_output: Fichier JSON généré par l'agent 2
    user_prompt: Prompt utilisateur original
    --model: Modèle LLM à utiliser (défaut: gemma3:27b)
    --backend: Backend LLM ('ollama' ou 'gemini', défaut: 'ollama')
"""

import argparse
import json
import logging
import os
import sys
import sys
import base64
from datetime import datetime
import markdown
from jinja2 import Environment, FileSystemLoader
from weasyprint import HTML, CSS
import re
import pandas as pd
import io

# Importation conditionnelle pour gérer l'absence de modules
DOCX_AVAILABLE = False
try:
    from docx import Document
    from docx.shared import Inches, Pt, RGBColor
    from docx.enum.text import WD_ALIGN_PARAGRAPH
    import html2text
    DOCX_AVAILABLE = True
except ImportError as e:
    logging.warning(f"Module python-docx non disponible. La génération de document Word sera désactivée.")
    logging.warning("Pour activer cette fonctionnalité, exécutez: pip install python-docx ht

# Importation du module llm_utils
from llm_utils import call_llm

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("agent3.log"),
        logging.StreamHandler(sys.stderr)
    ]
)
```

```

)
logger = logging.getLogger("agent3")

# =====
# Fonctions utilitaires pour les templates
# =====

def basename_filter(path):
    """
    Filtre Jinja2 pour obtenir le nom de base d'un chemin.

    Args:
        path: Chemin à traiter

    Returns:
        str: Nom de base du chemin
    """
    return os.path.basename(path) if path else path

def preprocess_markdown(text):
    """
    Prétraite le texte Markdown pour assurer une meilleure compatibilité avec les convertisseurs.

    Args:
        text: Texte Markdown à prétraiter

    Returns:
        str: Texte Markdown prétraité
    """
    if not text:
        return ""

    # Diviser le texte en lignes
    lines = text.split('\n')
    processed_lines = []

    in_list = False
    list_indent = 0

    for i, line in enumerate(lines):
        stripped = line.strip()

        # Correction des titres - s'assurer qu'il y a un espace après le #
        if stripped and stripped[0] == '#':

            for level in range(6, 0, -1): # Vérifier les niveaux de titre de h6 à h1
                tag = '#' * level
                if stripped.startswith(tag) and (len(stripped) == level or stripped[level] != ' '):
                    line = tag + ' ' + stripped[level:].lstrip()
                    break

            # Pré-traitement spécial pour les titres générés par le LLM
            # Souvent, le LLM peut générer des titres comme "# 1. TITRE" ou "## Titre"
            # sans que ce soit correctement interprété comme Markdown
            if stripped.startswith('# ') or stripped.startswith('## ') or stripped.startswith('### '):
                # Assurons-nous que ces titres soient sans symboles #
                level = 0
                while stripped[level] == '#':
                    level += 1
                line = stripped[level:].strip() # Supprime les # et les espaces

        # Correction des listes à puces
        if stripped.startswith('- ') and not stripped.startswith('- - '):

```

```

        spaces_before = len(line) - len(line.lstrip())
        line = ' ' * spaces_before + '- ' + stripped[1:].lstrip()
        in_list = True
        list_indent = spaces_before

# Correction des listes numériques
elif re.match(r'^\d+\.\S', stripped):
    num, rest = re.match(r'^(\d+\.)\S.*', stripped).groups()
    spaces_before = len(line) - len(line.lstrip())
    line = ' ' * spaces_before + num + ' ' + rest
    in_list = True
    list_indent = spaces_before

# Gestion des éléments de liste qui continuent sur plusieurs lignes
elif in_list and stripped and lines[i-1].strip() and len(line) - len(line.lstrip()):
    # C'est la continuation d'un élément de liste
    pass
elif stripped:
    # Ligne non vide qui n'est pas une continuation de liste
    in_list = False

# Correction des tableaux mal formatés
if '|' in line:
    # Assurer que les cellules de tableau ont des espaces appropriés
    parts = line.split('|')
    line = '|'.join([p.strip() for p in parts])

processed_lines.append(line)

return '\n'.join(processed_lines)

def markdown_filter(text):
    """
    Filtre Jinja2 amélioré pour convertir du Markdown en HTML avec support avancé.

    Args:
        text: Texte Markdown à convertir

    Returns:
        str: HTML généré
    """
    if not text:
        return ""

    try:
        # Prétraitement du texte Markdown
        text = preprocess_markdown(text)

        # Nettoyer les caractères # qui ne sont pas des titres (au milieu du texte)
        # Mais conserver ceux qui sont au début des lignes pour les convertir en titres
        lines = text.split('\n')
        for i, line in enumerate(lines):
            # Si le caractère # n'est pas au début de la ligne après des espaces, l'échapper
            if '#' in line and not line.lstrip().startswith('#'):
                # Remplacer les # qui ne sont pas au début de la ligne par \#
                parts = []
                j = 0
                while j < len(line):
                    if line[j] == '#' and (j == 0 or line[j-1] != '\\'):
                        # Vérifier si ce # est au début d'une ligne ou après des espaces
                        prefix = line[:j].rstrip()
                        if prefix: # S'il y a du texte avant, échapper le #
                            parts.append(prefix + '\\#')
                        parts.append(line[j:])
                        break
                    else:
                        parts.append(line[j])
                        j += 1
                line = ''.join(parts)
        text = '\n'.join(lines)
    
```

```

        j += 1
    if parts:
        lines[i] = ''.join(parts)

text = '\n'.join(lines)

# Utilisation d'extensions supplémentaires pour améliorer le rendu
html = markdown.markdown(text, extensions=[
    'fenced_code',      # Pour les blocs de code ```
    'codehilite',       # Coloration syntaxique des blocs de code
    'tables',           # Pour les tableaux
    'nl2br',            # Conversion des sauts de ligne
    'sane_lists',       # Listes mieux formatées
    'smarty',           # Guillemets intelligents et tirets
    'attr_list',         # Attributs pour les éléments
    'def_list',          # Listes de définition
    'footnotes',        # Notes de bas de page

    'md_in_html'        # Permet le Markdown dans les balises HTML
])

# Post-traitement pour corriger certains problèmes courants
# Assurer que les listes ont les bonnes classes CSS
html = html.replace('<ul>', '<ul class="report-list">')
html = html.replace('<ol>', '<ol class="report-list">')

# Assurer que les paragraphes dans les éléments de liste sont bien formatés
html = html.replace('<li><p>', '<li>')
html = html.replace('</p></li>', '</li>')

return html
except Exception as e:
    logger.error(f"Erreur lors de la conversion Markdown: {e}")
    # Fallback simple en cas d'erreur
    return f"<p>{text}</p>"

def csv_to_html_table(csv_data, max_rows=10):
    """
    Convertit les données CSV en tableau HTML.

    Args:
        csv_data: Données au format CSV sous forme de chaîne
        max_rows: Nombre maximum de lignes à afficher

    Returns:
        str: Tableau HTML ou message d'erreur
    """
    if not csv_data or len(csv_data.strip()) == 0:
        return ""

    try:
        # Parser les données CSV
        df = pd.read_csv(io.StringIO(csv_data))

        # Limiter le nombre de lignes
        if len(df) > max_rows:
            df = df.head(max_rows)
            footer = f"<p><em>Affichage des {max_rows} premières lignes sur {len(df)} au tot"
        else:
            footer = ""

        # Convertir en HTML
        table_html = df.to_html(index=False, classes="csv-data-table")

```

```

        return f"<div class='csv-data-container'>{table_html}{footer}</div>"

    except Exception as e:
        logger.error(f"Erreur lors de la conversion des données CSV en HTML: {e}")
        return f"<div class='error'>Erreur de conversion des données CSV: {str(e)}</div>"

def parse_csv_for_summary(csv_data):
    """
    Extrait des informations résumées des données CSV.

    Args:
        csv_data: Données au format CSV sous forme de chaîne

    Returns:
        dict: Résumé des données ou dictionnaire vide en cas d'erreur
    """
    if not csv_data or len(csv_data.strip()) == 0:
        return {}

    try:
        # Parser les données CSV
        df = pd.read_csv(io.StringIO(csv_data))

        # Créer un résumé
        summary = {
            "num_rows": len(df),
            "num_cols": len(df.columns),
            "columns": list(df.columns),
            "numeric_stats": {}
        }

        # Ajouter des statistiques pour les colonnes numériques
        for col in df.select_dtypes(include=['number']).columns:
            summary["numeric_stats"][col] = {
                "min": df[col].min(),
                "max": df[col].max(),
                "mean": df[col].mean(),
                "median": df[col].median(),
            }

        return summary

    except Exception as e:
        logger.error(f"Erreur lors de l'analyse CSV pour résumé: {e}")
        return {}

# =====
# Fonctions de traitement des images et visualisations
# =====

def save_images(visualisations, img_dir):
    """
    Sauvegarde les visualisations à partir de base64 et retourne une liste de dictionnaires.
    Prend en charge à la fois les visualisations et les tables de régression.

    Args:
        visualisations: Liste des métadonnées des visualisations
        img_dir: Répertoire où sauvegarder les images

    Returns:
        Liste des informations sur les images sauvegardées
    """
    os.makedirs(img_dir, exist_ok=True)

```

```

image_infos = [] # Pour stocker plus d'infos

for i, vis in enumerate(visualisations):
    # Déterminer l'identifiant et le titre
    if 'filename' in vis:
        base_filename = vis.get("filename", f"figure_{i+1}.png")
        vis_id = os.path.splitext(base_filename)[0]
    else:
        vis_id = vis.get("id", f"item_{i+1}")
        base_filename = vis_id + '.png'

    # Déterminer le type de visualisation
    vis_type = "visualisation"
    if 'type' in vis and vis['type'] == 'regression_table':
        vis_type = "regression_table"

    # Déterminer le titre
    title = vis.get("title", vis_id.replace('_', ' ').capitalize())

    # Nom de fichier standardisé
    img_filename = f"{vis_id}.png"
    img_path = os.path.join(img_dir, img_filename)

    if "base64" in vis:
        try:
            img_data = base64.b64decode(vis["base64"])
            with open(img_path, "wb") as f:
                f.write(img_data)

            # Journaliser plus d'informations pour le débogage
            file_size = os.path.getsize(img_path)
            logger.info(f"Sauvegarde de l'image {img_path} avec taille: {file_size} octets")

            # Extraire les données CSV si disponibles
            csv_data = vis.get("csv_data", "")
            csv_summary = parse_csv_for_summary(csv_data) if csv_data else {}

            # Extraire l'interprétation détaillée si disponible (pour les régressions)
            detailed_interpretation = vis.get("detailed_interpretation", "")

            # Créer l'objet d'information avec les métadonnées
            image_info = {
                "filename": img_filename,
                "path": img_path,
                "title": title,
                "type": vis_type,
                "interpretation": vis.get("interpretation", ""),
                "detailed_interpretation": detailed_interpretation, # Ajouter l'interprétation
                "metadata": vis.get("metadata", {}),
                "data": vis.get("data", {}),
                "csv_data": csv_data,
                "csv_summary": csv_summary,
                "csv_html": csv_to_html_table(csv_data) if csv_data else "",
                "size": file_size
            }

            image_infos.append(image_info)
            logger.info(f"Image sauvegardée: {img_path} avec titre: {image_info['title']}")
            if csv_data:
                logger.info(f" -> Données CSV trouvées pour: {img_filename} ({len(csv_data)} lignes)")
            if detailed_interpretation:
                logger.info(f" -> Interprétation détaillée de {len(detailed_interpretation)} caractères")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde de l'image {vis_id}: {e}")
            logger.error(f"Contenu de 'base64': {vis.get('base64', '')[:50]}...")

```

```

else:
    logger.warning(f"Pas de données base64 pour l'image {vis_id}")
    # Si path est présent, essayer de copier l'image
    if 'path' in vis and os.path.exists(vis['path']):
        try:
            import shutil
            shutil.copy(vis['path'], img_path)

            # Extraire les données CSV si disponibles
            csv_data = vis.get("csv_data", "")
            csv_summary = parse_csv_for_summary(csv_data) if csv_data else {}

            # Extraire l'interprétation détaillée (pour les régressions)
            detailed_interpretation = vis.get("detailed_interpretation", "")

            image_info = {
                "filename": img_filename,
                "path": img_path,
                "title": title,
                "type": vis_type,
                "interpretation": vis.get("interpretation", ""),
                "detailed_interpretation": detailed_interpretation, # Ajouter l'int
                "metadata": vis.get("metadata", {}),
                "data": vis.get("data", {}),
                "csv_data": csv_data,
                "csv_summary": csv_summary,

                "csv_html": csv_to_html_table(csv_data) if csv_data else ""
            }
            image_infos.append(image_info)
            logger.info(f"Image copiée depuis {vis['path']} vers {img_path}")
            if detailed_interpretation:
                logger.info(f" -> Interprétation détaillée de {len(detailed_interpr
        except Exception as e:
            logger.error(f"Erreur lors de la copie de l'image {vis['path']}: {e}")

    return image_infos

# =====
# Fonctions pour la mise en forme des résultats de régression
# =====

def format_regression_results(regression_text):
    """
    Formate les résultats de régression pour une meilleure présentation.

    Args:
        regression_text: Texte brut des résultats de régression

    Returns:
        str: HTML formaté des résultats
    """
    # Vérifier si c'est un résultat de régression OLS
    if "OLS Regression Results" not in regression_text:
        return f"<pre class='results-block'>{regression_text}</pre>"

    # Extraire les sections principales
    sections = {}

    # Extraire R-squared et Adj. R-squared
    r_squared_match = re.search(r"R-squared:\s+([\d\.]+)", regression_text)
    adj_r_squared_match = re.search(r"Adj. R-squared:\s+([\d\.]+)", regression_text)

```

```

sections['r_squared'] = r_squared_match.group(1) if r_squared_match else "N/A"
sections['adj_r_squared'] = adj_r_squared_match.group(1) if adj_r_squared_match else "N/A"

# Extraire les coefficients et statistiques
coef_section = re.search(r"==+\s*\n\s*(coef.*?)\n(?:\s*\n\s*==+|\Z)", regression_text, re.DOTALL)

# Construire un tableau HTML formaté
html = '<div class="regression-formatted">'
html += f'<h4>Résultats de la Régression</h4>'

# Ajouter les métriques clés
html += '<div class="regression-metrics">'
html += f'<span class="metric"><strong>R²:</strong> {sections["r_squared"]}</span>'
html += f'<span class="metric"><strong>R² ajusté:</strong> {sections["adj_r_squared"]}</span>'
html += '</div>'

# Ajouter le tableau des coefficients si disponible
if coef_section:
    html += '<table class="regression-coefficients">'

    # Créer l'en-tête du tableau
    html += '<thead><tr>'
    headers = re.findall(r"(\w+(?:\s+\w+)*)", coef_section.group(1).split('\n')[0])
    for header in headers:
        html += f'<th>{header}</th>'
    html += '</tr></thead>'

    # Ajouter les lignes de coefficients
    html += '<tbody>'

    lines = coef_section.group(1).split('\n')[1:] # Skip the header line
    for line in lines:
        if not line.strip():
            continue

        html += '<tr>'
        cells = re.findall(r"([\w\.\-]+(?:\s+[\w\.\-]+)*)", line)
        for i, cell in enumerate(cells):
            if i == 0: # Variable name
                html += f'<td><strong>{cell}</strong></td>'
            else:
                html += f'<td>{cell}</td>'
        html += '</tr>'

    html += '</tbody></table>'

html += '</div>'

return html

# Function to parse regression tables
def parse_regression_to_html(narrative_text, regression_tables):
    """
    Tente de trouver une table OLS dans le texte et la convertit en HTML <table>.
    Utilise également les tables de régression capturées si disponibles.

    Args:
        narrative_text: Texte contenant potentiellement des tables de régression
        regression_tables: Tables de régression capturées

    Returns:
        str: HTML des tables de régression ou None si non trouvé/erreur
    """
    # S'il y a des tables de régression capturées, les utiliser prioritairement

```

```

if regression_tables and len(regression_tables) > 0:
    logger.info(f"Utilisation de {len(regression_tables)} tables de régression capturées")

    # Créer un tableau HTML pour chaque table de régression
    html_tables = []

    for i, table in enumerate(regression_tables):
        title = table.get('title', f'Régression {i+1}')
        r_squared = table.get('metadata', {}).get('r_squared', 'N/A')

        html = f"<div class='regression-container'>\n"
        html += f"<h4>{title}</h4>\n"

        # Utiliser l'image de la table
        img_path = table.get('path')
        if img_path and os.path.exists(img_path):
            # Utiliser un chemin relatif basé sur la position du fichier HTML
            rel_path = os.path.basename(img_path)
            html += f"<img src='images/{rel_path}' alt='{title}' class='regression-image'"
            logger.info(f"Image de régression ajoutée: images/{rel_path}")
        else:
            logger.warning(f"Chemin d'image non valide pour {title}: {img_path}")

        # Ajouter l'interprétation détaillée si disponible
        detailed_interpretation = table.get('detailed_interpretation', '')
        if detailed_interpretation:
            html += f"<div class='regression-interpretation detailed'>\n"
            html += f"<h5>Interprétation économétrique détaillée</h5>\n"
            html += f"{markdown_filter(detailed_interpretation)}\n"
            html += f"</div>\n"
            logger.info(f"Interprétation détaillée de la régression ajoutée pour {title}")
        # Sinon, utiliser l'interprétation standard si disponible
        elif 'interpretation' in table and table['interpretation']:
            html += f"<div class='regression-interpretation'>\n"
            html += f"<h5>Interprétation</h5>\n"
            html += f"{markdown_filter(table['interpretation'])}\n"
            html += f"</div>\n"

        # Ajouter les données CSV si disponibles
        csv_data = table.get('csv_data', '')
        if csv_data:
            csv_html = csv_to_html_table(csv_data)
            if csv_html:
                html += f"<div class='regression-data-table'>\n<h5>Données de la régression</h5>\n{csv_html}</div>\n"

        # Ajouter les données structurées si disponibles
        if 'data' in table and table['data']:
            coefficients_data = table['data'].get('coefficients', [])
            if coefficients_data:
                html += "<div class='regression-data'>\n"
                html += "<h5>Coefficients significatifs</h5>\n"
                html += "<table class='coefficients-table'>\n"
                html += "<thead><tr><th>Variable</th><th>Coefficient</th><th>Valeur p</th></tr>\n"
                html += "<tbody>\n"

                for coef in coefficients_data:
                    var_name = coef.get('variable', 'N/A')
                    coef_value = coef.get('coef', 'N/A')
                    p_value = coef.get('p_value', 'N/A')

                    # Marquer les coefficients significatifs
                    if p_value != 'N/A':
                        try:

```

```

        p_value_float = float(p_value)
        is_significant = p_value_float < 0.05
        tr_class = "significant" if is_significant else ""
        significance_text = "Significatif" if is_significant else "Non significatif"
        html += f"<tr class='{tr_class}'>"
    except ValueError:
        html += "<tr>"
        significance_text = "Indéterminé"
    else:
        html += "<tr>"
        significance_text = "Indéterminé"

    html += f"<td>{var_name}</td><td>{coef_value}</td><td>{p_value}</td>"

    html += "</tbody></table>\n"
    html += "</div>\n"

    html += "</div>\n"
    html_tables.append(html)

return "\n".join(html_tables)

# Pour la recherche dans le texte narrative
match = re.search(r"={10,}\s*\n\s*OLS Regression Results\s*\n={10,}(.*)\n={10,}", narra

if match:
    logger.info("Table de régression OLS détectée dans le texte narrative.")
    table_content = match.group(1).strip()

    # Utiliser la fonction de formatage améliorée
    return format_regression_results("OLS Regression Results\n" + table_content)

# Fallback : chercher dans le texte narrative
match = re.search(r"OLS Regression Results\s*\n={10,}(.*)\n={10,}", narrative_text, re.

if match:
    logger.info("Table de régression OLS détectée dans le texte narrative.")

    table_content = match.group(1).strip()
    lines = table_content.split('\n')

    html = "<table class='regression-table'>\n"
    in_header_section = True # Les premières lignes sont souvent des clés:valeurs

    for line in lines:
        stripped_line = line.strip()
        if not stripped_line: continue # Ignore les lignes vides

        # Détecter la ligne d'en-tête du tableau principal (ex: coef std err t P>|t| [0.
        # Ceci est une heuristique et pourrait nécessiter un ajustement
        if re.match(r"^s*coef\s+std err\s+t\s+P>\|t\|", stripped_line, re.IGNORECASE):
            in_header_section = False
            # Ajouter la ligne d'en-tête comme <thead>
            headers = re.split(r'\s{2,}', stripped_line) # Split sur 2+ espaces
            html += " <thead>\n <tr><th></th>" # Colonne pour le nom de la variable
            for header in headers:
                html += f"<th>{header.strip()}</th>"
            html += "</tr>\n </thead>\n <tbody>\n"
            continue # Passer à la ligne suivante
        elif stripped_line.startswith('---'): # Ligne de séparation
            continue

    if in_header_section:

```

```

        # Section clé:valeur (ex: R-squared: 0.950)
        parts = stripped_line.split(':', 1)
        if len(parts) == 2:
            html += f" <tr><td style='font-weight:bold;'>{parts[0].strip()}</td><td>{parts[1].strip()}</td></tr>\n"
        else: # Ligne qui n'est pas clé:valeur
            html += f" <tr><td colspan='99'>{stripped_line}</td></tr>\n"
    else:
        # Lignes de données du tableau principal
        # La première colonne est souvent le nom de la variable, le reste sont des chiffres
        row_data = re.split(r'\s{2,}', stripped_line)
        html += " <tr>\n"
        for i, cell in enumerate(row_data):
            style = "text-align: right;" if i > 0 else "" # Aligner les chiffres à droite
            html += f" <td style='{style}'>{cell.strip()}</td>\n"
        html += " </tr>\n"

    if not in_header_section: # S'il y avait un corps de tableau
        html += " </tbody>\n"
    html += "</table>\n"
    return html
else:
    logger.info("Aucune table de régression OLS formatée détectée pour conversion HTML.")
    # Fallback: Mettre tout le bloc dans <pre> si on trouve "OLS Regression Results" mais pas de table
    ols_match = re.search(r"(OLS Regression Results[\s\S]*)", narrative_text, re.IGNORECASE)
    if ols_match:
        logger.warning("Table OLS trouvée mais format non reconnu pour <table>. Utiliser <pre> à la place.")
        return f"<pre class='regression-text'>{ols_match.group(1).strip()}</pre>"
    return None

# =====
# Fonctions de génération de contenu académique
# =====

def generate_comprehensive_economic_analysis(agent1_data, agent2_data, model, backend):
    """
    Génère un raisonnement économique complet en agrégeant toutes les interprétations
    des visualisations, tables OLS et la problématique initiale.

    Args:
        agent1_data: Données de l'agent1
        agent2_data: Données de l'agent2
        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM

    Returns:
        str: Raisonnement économique complet
    """
    # Vérifier si un modèle puissant a été utilisé par l'agent 2
    if "current_model" in agent2_data and agent2_data["current_model"] != model:
        logger.info(f"Utilisation du modèle puissant transmis par l'agent 2: {agent2_data['current_model']}")
        model = agent2_data["current_model"]

    # Récupérer la problématique (prompt utilisateur)
    user_prompt = agent1_data.get("user_prompt", "")

    # Récupérer les sections académiques de l'agent 1
    introduction = agent1_data.get('llm_output', {}).get('introduction', '')
    hypotheses = agent1_data.get('llm_output', {}).get('hypotheses', '')
    literature_review = agent1_data.get('llm_output', {}).get('literature_review', '')
    methodology = agent1_data.get('llm_output', {}).get('methodology', '')
    limitations = agent1_data.get('llm_output', {}).get('limitations', '')

    # Recueillir toutes les interprétations des visualisations
    all_interpretations = []
    if "visualisations" in agent2_data:

```

```

        for i, vis in enumerate(agent2_data["visualisations"]):
            if 'interpretation' in vis and vis['interpretation']:
                title = vis.get('title', f'Visualisation {i+1}')
                interp = vis['interpretation']
                all_interpretations.append(f"### {title}\n{interp}")

# Joindre toutes les interprétations
interpretations_text = "\n\n".join(all_interpretations)

# Récupérer la narration de l'agent 2

narrative = agent2_data.get("narrative", "")

# Créer le prompt pour le raisonnement économique complet
prompt = f"### Raisonnement économique complet et approfondi

### Problématique de recherche
{user_prompt}

### Contextualisation académique
{introduction[:500]}...

### Hypothèses de recherche
{hypotheses}

### Revue de littérature
{literature_review[:500]}...

### Méthodologie appliquée
{methodology[:500]}...

### Résultats techniques
{narrative[:1000]}...

### Interprétations des visualisations et régressions
{interpretations_text}

---

```

À partir des éléments ci-dessus, produisez un raisonnement économique complet et approfondi

1. Synthèse globale
  - Résumez l'objectif et le contexte global de l'étude
  - Rappeler les principales hypothèses testées
  - Résumez les observations empiriques clés
2. Analyse économique approfondie
  - Interprétez les résultats dans un cadre économique rigoureux
  - Identifiez les mécanismes économiques sous-jacents
  - Expliquez les relations causales et corrélations observées
  - Reliez explicitement les résultats aux théories économiques pertinentes
  - Discutez les implications économiques des coefficients significatifs
3. Limites et nuances
  - Examinez la validité interne et externe des résultats
  - Discutez des biais potentiels et de leurs impacts sur l'interprétation
  - Suggérez des perspectives alternatives d'interprétation
4. Implications pratiques et théoriques
  - Formulez des recommandations concrètes pour les décideurs ou acteurs économiques
  - Identifiez les contributions théoriques à la littérature académique

Cette analyse doit être rigoureuse, nuancée et suffisamment développée pour capturer la comp

IMPORTANT:

- Utilisez un format de titre normal (pas de majuscules) et sans pourcentages
  - Formatez clairement les points 1, 2, 3, et 4 comme des sections avec des titres
  - Structurez les sous-points avec des listes à puces ou des sous-titres
  - Utilisez une formulation claire et professionnelle
  - Réponse en Français
- """

```
try:
    logger.info(f"Génération du raisonnement économique complet avec le modèle {model}")
    analysis = call_llm(prompt=prompt, model_name=model, backend=backend)

    # Nettoyage supplémentaire pour supprimer les symboles # des titres
    lines = analysis.split('\n')
    cleaned_lines = []

    for line in lines:
        stripped = line.strip()
        # Si la ligne est un titre (commence par # suivi d'espace), supprimer les #
        if stripped.startswith('# ') or stripped.startswith('## ') or stripped.startswith('### '):
            # Compter le nombre de #
            level = 0
            while level < len(stripped) and stripped[level] == '#':
                level += 1

            # Extraire le titre sans les #
            title_text = stripped[level:].strip()

            # Ajouter le titre formaté selon son niveau
            if level == 1:
                cleaned_lines.append(f"Synthèse globale")
            elif level == 2:
                cleaned_lines.append(f"Analyse économique approfondie")
            elif level == 3:
                cleaned_lines.append(f"Limites et nuances")
            elif level == 4:
                cleaned_lines.append(f"Implications pratiques et théoriques")
            else:
                cleaned_lines.append(title_text)
        else:
            cleaned_lines.append(line)

    return '\n'.join(cleaned_lines)
except Exception as e:
    logger.error(f"Erreur lors de la génération du raisonnement économique: {e}")
    return "Erreur lors de la génération du raisonnement économique complet."
```

```
def generate_comprehensive_visual_analysis(visualizations, agent1_data, agent2_data, model,
    """
```

Génère une analyse globale de toutes les visualisations.

Args:

visualizations: Liste des métadonnées des visualisations avec interprétations  
agent1\_data: Données de l'agent1  
agent2\_data: Données de l'agent2  
model: Modèle LLM à utiliser  
backend: Backend pour les appels LLM

Returns:

str: Analyse globale des visualisations

"""

# Vérifier si un modèle puissant a été utilisé par l'agent 2

if "current\_model" in agent2\_data and agent2\_data["current\_model"] != model:

```

        logger.info(f"Utilisation du modèle puissant transmis par l'agent 2: {agent2_data['current_model']}")
        model = agent2_data["current_model"]

# Séparer les visualisations des tables de régression
charts = [v for v in visualizations if v.get('type') != 'regression_table']
regression_tables = [v for v in visualizations if v.get('type') == 'regression_table']

# Extract visualization titles
vis_descriptions = []
for i, vis in enumerate(charts):
    title = vis.get('title', f'Figure {i+1}')
    vis_descriptions.append(f"***{title}***")

# Extract regression table information
regression_descriptions = []
for i, table in enumerate(regression_tables):
    title = table.get('title', f'Regression Table {i+1}')
    r_squared = table.get('metadata', {}).get('r_squared', 'N/A')
    variables = table.get('metadata', {}).get('variables', [])
    regression_descriptions.append(f"***{title}** (R² = {r_squared}, Variables: {'', '.join(variables)})")

# Join visualization descriptions
vis_descriptions_text = '\n'.join(vis_descriptions)
regression_descriptions_text = '\n'.join(regression_descriptions)

# Récupérer les interprétations générées par Gemini Flash
interp_descriptions = []
for i, vis in enumerate(visualizations):
    title = vis.get('title', f'Item {i+1}')
    interp = vis.get('interpretation', 'Interprétation non disponible')
    interp_descriptions.append(f"### {title}\n{interp}")

interp_text = '\n\n'.join(interp_descriptions)

# Extraire les éléments importants des agents 1 et 2
introduction = agent1_data.get('llm_output', {}).get('introduction', 'Non disponible')
hypotheses = agent1_data.get('llm_output', {}).get('hypotheses', 'Non disponible')
methodology = agent1_data.get('llm_output', {}).get('methodology', 'Non disponible')
narrative = agent2_data.get("narrative", "Non disponible")

# Créer le prompt pour la synthèse
prompt = f"### ANALYSE GLOBALE DES VISUALISATIONS ÉCONOMIQUES

### Question de recherche initiale
{agent1_data.get("user_prompt", "Non disponible")}

### Conceptualisation académique (Agent 1)
Introduction: {introduction[:500]}...
Hypothèses: {hypotheses[:500]}...
Méthodologie: {methodology[:300]}...

### Résultats d'analyse (Agent 2)
Narration: {narrative[:500]}...

### Visualisations à interpréter
{vis_descriptions_text}

### Tables de régression
{regression_descriptions_text}

### Interprétations détaillées existantes
{interp_text[:3000]}

---
```

Générez une analyse globale CONCISE (250-300 mots maximum) qui:

1. Fait la synthèse des tendances clés observées dans les visualisations
2. Relie directement chaque observation aux hypothèses formulées
3. Se concentre uniquement sur les relations économiques les plus significatives
4. Évite toute description technique au profit d'une interprétation économique claire
5. Utilise un langage simple, factuel et direct
6. En français

Cette analyse doit être une vue d'ensemble cohérente et condensée des résultats principaux,

IMPORTANT: Structurez votre réponse en paragraphes clairs et utilisez des transitions logiques

```
try:
    logger.info(f"Génération de l'analyse visuelle globale avec le modèle {model}")
    content = call_llm(prompt=prompt, model_name=model, backend=backend)

    # Nettoyer les caractères # qui pourraient se trouver dans le texte

    lines = content.split('\n')
    cleaned_lines = []

    for line in lines:
        if line.strip().startswith('#'):
            # Nettoyer les symboles # au début des lignes
            cleaned_line = re.sub(r'^#+ ', '', line)
            cleaned_lines.append(cleaned_line)
        else:
            cleaned_lines.append(line)

    return '\n'.join(cleaned_lines)
except Exception as e:
    logger.error(f"Erreur lors de la génération de l'analyse visuelle globale: {e}")
    return "Erreur lors de la génération de l'analyse visuelle globale."

def generate_synthesis(agent1_data, agent2_data, model, backend):
    """
    Génère une synthèse de type résumé simple.

    Args:
        agent1_data: Données de l'agent1
        agent2_data: Données de l'agent2
        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM

    Returns:
        str: Résumé simple
    """
    # Vérifier si un modèle puissant a été utilisé par l'agent 2
    if "current_model" in agent2_data and agent2_data["current_model"] != model:
        logger.info(f"Utilisation du modèle puissant transmis par l'agent 2: {agent2_data['current_model']}")
        model = agent2_data["current_model"]

    # Récupération des données importantes
    narrative = agent2_data.get("narrative", "Aucune narration disponible")

    # Récupération des données académiques de l'agent1 si disponibles
    introduction = agent1_data.get('llm_output', {}).get('introduction', '')
    hypotheses = agent1_data.get('llm_output', {}).get('hypotheses', '')
    methodology = agent1_data.get('llm_output', {}).get('methodology', '')

    # Informations sur les visualisations pour enrichir la synthèse
```

```

        visualisations_info = ""
        if "visualisations" in agent2_data:
            vis_count = len([v for v in agent2_data["visualisations"] if v.get('type') != 'regre
            reg_count = len([v for v in agent2_data["visualisations"] if v.get('type') == 'regre
            visualisations_info = f"L'analyse contient {vis_count} visualisations graphiques et

    prompt = f"### GÉNÉRATION DE RÉSUMÉ SIMPLE

### Narration détaillée de l'analyse
{narrative[:1500]}

### Contextualisation et objectifs de recherche
{introduction[:500]}

### Hypothèses de recherche
{hypotheses[:300]}

### Méthodologie appliquée
{methodology[:300]}

### Informations sur les visualisations
{visualisations_info}

---

Rédigez un résumé très concis des analyses effectuées 150 mots environ. Cette synthèse doit

1. L'objectif principal de l'analyse (1 phrase)
2. Les principales méthodes utilisées, en termes simples (1 phrase)
3. Les résultats les plus importants (2-3 phrases)
4. La conclusion principale (1 phrase)

IMPORTANT:
- Utilisez un langage simple, direct et clair
- Évitez tout jargon technique
- Structurez le texte en un seul paragraphe sans puces ni énumérations
"""

    try:
        logger.info(f"Appel LLM via backend '{backend}' avec modèle '{model}' pour génératio
        content = call_llm(prompt=prompt, model_name=model, backend=backend)

        # Nettoyer les caractères # qui pourraient se trouver dans le texte
        content = content.replace('# ', '').replace('## ', '').replace('### ', '')

        return content
    except Exception as e:
        logger.error(f"Erreur lors de la génération du résumé: {e}")
        return f"Erreur lors de la génération du résumé: {e}"

def generate_discussion_section(agent1_data, agent2_data, model, backend):
    """
    Génère une section discussion simple et accessible.

    Args:
        agent1_data: Données de l'agent1
        agent2_data: Données de l'agent2

        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM

    Returns:
        str: Section discussion

```

```

"""
# Vérifier si un modèle puissant a été utilisé par l'agent 2
if "current_model" in agent2_data and agent2_data["current_model"] != model:
    logger.info(f"Utilisation du modèle puissant transmis par l'agent 2: {agent2_data['c
    model = agent2_data["current_model"]

narrative = agent2_data.get("narrative", "Aucune narration disponible")
hypotheses = agent1_data.get('llm_output', {}).get('hypotheses', '')
limitations = agent1_data.get('llm_output', {}).get('limitations', '')
literature_review = agent1_data.get('llm_output', {}).get('literature_review', '')

# Extraire des informations sur les résultats de régression si disponibles
regression_info = ""

if "visualisations" in agent2_data:
    for vis in agent2_data["visualisations"]:
        if vis.get("type") == "regression_table":
            r_squared = vis.get("metadata", {}).get("r_squared", "N/A")
            variables = vis.get("metadata", {}).get("variables", [])
            regression_info += f"Régression avec R² = {r_squared}, variables clés: {'', '

            # Extraire les interprétations générées par Gemini Flash
            interpretation = vis.get("interpretation", "")
            if interpretation:
                regression_info += f"Interprétation par Gemini Flash: {interpretation[:2

# Récupérer quelques interprétations de visualisations pour enrichir la discussion
vis_interpretations = []
if "visualisations" in agent2_data:
    for vis in agent2_data["visualisations"]:
        if vis.get("type") != "regression_table":
            interpretation = vis.get("interpretation", "")
            if interpretation:
                title = vis.get("title", "Figure")
                vis_interpretations.append(f"### {title}\n{interpretation[:200]}...")
                if len(vis_interpretations) >= 2: # Limiter à 2 interprétations
                    break

vis_interpretations_text = "\n\n".join(vis_interpretations)

prompt = f"""### GÉNÉRATION DE SECTION DISCUSSION CONCISE

### Résultats de l'analyse
{narrative[:1000]}

### Interprétations des visualisations
{vis_interpretations_text}

### Informations sur les régressions
{regression_info}

### Hypothèses de recherche initiales
{hypotheses[:300]}

### Limitations méthodologiques
{limitations[:300]}

---

Rédigez une section "Discussion" concise (250-300 mots maximum) qui interprète les résultats

STRUCTURE ESSENTIELLE:
1. **Interprétation des résultats principaux** (1 paragraphe)
    - Expliquez ce que signifient les résultats les plus importants
    - Mettez en évidence les relations les plus significatives

```

2. **\*\*Limites et précautions\*\*** (1 paragraphe court)
  - Mentionnez 2-3 limites principales de l'étude
  - Expliquez brièvement leur impact sur l'interprétation
3. **\*\*Implications pratiques\*\*** (1 paragraphe court)
  - Discutez des applications concrètes de ces résultats

#### EXIGENCES STYLISTIQUES:

- Langage simple et direct, sans jargon technique
- Paragraphes courts (3-4 phrases maximum)
- Ne dépassez pas 300 mots au total
- Ton neutre et factuel
- Utilisez des connecteurs logiques clairs entre les paragraphes pour assurer une progression

```

try:
    logger.info(f"Appel LLM via backend '{backend}' avec modèle '{model}' pour génération")
    content = call_llm(prompt=prompt, model_name=model, backend=backend)

    # Nettoyer les caractères # qui pourraient se trouver dans le texte
    lines = content.split('\n')
    cleaned_lines = []

    for line in lines:
        if line.strip().startswith('#'):
            # Nettoyer les symboles # au début des lignes
            cleaned_line = re.sub(r'^#+ ', '', line)
            cleaned_lines.append(cleaned_line)
        else:

            cleaned_lines.append(line)

    return '\n'.join(cleaned_lines)
except Exception as e:
    logger.error(f"Erreur lors de la génération de la discussion: {e}")
    return "Erreur lors de la génération de la section discussion."

def generate_conclusion_section(agent1_data, agent2_data, model, backend):
    """
    Génère une conclusion simple.

    Args:
        agent1_data: Données de l'agent1
        agent2_data: Données de l'agent2
        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM

    Returns:
        str: Section conclusion
    """
    # Vérifier si un modèle puissant a été utilisé par l'agent 2
    if "current_model" in agent2_data and agent2_data["current_model"] != model:
        logger.info(f"Utilisation du modèle puissant transmis par l'agent 2: {agent2_data['current_model']}")
        model = agent2_data["current_model"]

    narrative = agent2_data.get("narrative", "Aucune narration disponible")
    introduction = agent1_data.get('llm_output', {}).get('introduction', '')

    # Informations sur les visualisations
    vis_summary = ""
    if "visualisations" in agent2_data:
        vis_types = []
        regression_count = 0

```

```

for vis in agent2_data["visualisations"]:
    if vis.get("type") == "regression_table":
        regression_count += 1
    else:
        filename = vis.get("filename", "")
        if "correlation" in filename.lower():
            vis_types.append("matrices de corrélation")
        elif "scatter" in filename.lower() or "relation" in filename.lower():
            vis_types.append("nuages de points")
        elif "box" in filename.lower():
            vis_types.append("boîtes à moustaches")
        elif "histogram" in filename.lower() or "distribution" in filename.lower():
            vis_types.append("histogrammes")

vis_types = list(set(vis_types)) # Éliminer les doublons

if vis_types or regression_count > 0:
    vis_summary = "L'analyse a utilisé "
    if vis_types:
        vis_summary += f"des visualisations ({', '.join(vis_types)})"

    if regression_count > 0:
        if vis_types:
            vis_summary += f" et {regression_count} modèles de régression"
        else:
            vis_summary += f"{regression_count} modèles de régression"

    vis_summary += " pour explorer les données."

prompt = f"""## GÉNÉRATION DE CONCLUSION CONCISE

### Résultats de l'analyse
{narrative[:800]}

### Introduction et objectifs initiaux
{introduction[:300]}

### Résumé des visualisations
{vis_summary}

---

Rédigez une conclusion très concise (150 mots maximum) pour ce rapport d'analyse. Cette conc

STRUCTURE ESSENTIELLE:
1. **Rappel bref de l'objectif** (1 phrase)
2. **Résumé des 2-3 découvertes les plus importantes** (2-3 phrases)
3. **Implications pratiques principales** (1-2 phrases)
4. **Conclusion générale** (1 phrase d'ouverture sur des perspectives futures)

EXIGENCES:
- Concision extrême: 150 mots maximum
- Langage simple et direct
- Ton positif et constructif
- Pas de nouveaux éléments ou analyses
- Un seul paragraphe compact
- Assurez-vous que la conclusion s'inscrit dans la suite logique des autres sections
"""

try:
    logger.info(f"Appel LLM via backend '{backend}' avec modèle '{model}' pour génération")
    content = call_llm(prompt=prompt, model_name=model, backend=backend)

    # Nettoyer les caractères # qui pourraient se trouver dans le texte
    lines = content.split('\n')

```

```

        cleaned_lines = []

        for line in lines:
            if line.strip().startswith('#'):
                # Nettoyer les symboles # au début des lignes
                cleaned_line = re.sub(r'^#+ ', '', line)
                cleaned_lines.append(cleaned_line)
            else:
                cleaned_lines.append(line)

        return '\n'.join(cleaned_lines)
    except Exception as e:
        logger.error(f"Erreur lors de la génération de la conclusion: {e}")
        return "Erreur lors de la génération de la conclusion."

def generate_references(agent1_data, model, backend):
    """
    Génère des références bibliographiques simples.

    Args:
        agent1_data: Données de l'agent1
        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM

    Returns:
        str: Références bibliographiques
    """
    literature_review = agent1_data.get('llm_output', {}).get('literature_review', '')
    user_prompt = agent1_data.get('user_prompt', '')

    prompt = f"""## GÉNÉRATION DE RÉFÉRENCES SIMPLES

### Indications sur la littérature pertinente
{literature_review[:800]}

### Sujet de recherche
{user_prompt}

---

Générez une liste CONCISE de 4-5 références essentielles pour ce sujet d'analyse. Ces référe

EXIGENCES:
1. **Nombre et variété**:
    - 4-5 références pertinentes et accessibles
    - Privilégiez les sources récentes (après 2015)

2. **Format simple et compact**:
    - Format: Auteur(s), (année). Titre. Source.
    - Pas de formatage complexe
    - Classement par ordre alphabétique d'auteur

IMPORTANT:
- Les références doivent être RÉELLES et VÉRIFIABLES
- Privilégiez les références en français quand c'est possible
- Évitez les références trop spécialisées ou peu connues
- Utilisez des puces (format Markdown) pour présenter chaque référence
"""

    try:
        logger.info(f"Appel LLM via backend '{backend}' pour génération des références bibliographiques")
        content = call_llm(prompt=prompt, model_name=model, backend=backend)
    
```

```

# Nettoyer les caractères # qui pourraient se trouver dans le texte
lines = content.split('\n')
cleaned_lines = []

for line in lines:
    if line.strip().startswith('#'):
        # Nettoyer les symboles # au début des lignes
        cleaned_line = re.sub(r'^#+ ', '', line)
        cleaned_lines.append(cleaned_line)
    else:
        cleaned_lines.append(line)

    return '\n'.join(cleaned_lines)
except Exception as e:
    logger.error(f"Erreur lors de la génération des références: {e}")
    return "Erreur lors de la génération des références bibliographiques."

def interpret_visualization_with_gemini(vis, agent1_data, model, backend, prompts_log_path,
    """
    Interprète une visualisation en demandant une interprétation concise contextualisée avec

    Args:
        vis: Métadonnées de la visualisation avec base64 de l'image
        agent1_data: Données de l'agent1
        model: Modèle LLM à utiliser
        backend: Backend pour les appels LLM
        prompts_log_path: Chemin pour sauvegarder les prompts

    Returns:
        str: Interprétation de la visualisation
    """
    from llm_utils import call_llm

    # Valider que l'image est disponible
    if 'base64' not in vis or not vis['base64']:
        logger.error(f"Pas de données base64 disponibles pour la visualisation {vis.get('id', '')}")
        return "Erreur: Impossible d'analyser cette visualisation (image non disponible)"

    # Extraire le titre et le type
    if 'filename' in vis:
        filename = vis.get("filename", "figure.png")
        vis_id = os.path.splitext(filename)[0]
    else:
        vis_id = vis.get("id", "visualisation")
        filename = vis_id + '.png'

    # Déterminer le type de visualisation
    vis_type = "Visualisation"
    if 'regression' in vis_id.lower():
        vis_type = "Table de Régression OLS"
    elif 'correlation' in vis_id.lower() or 'corr' in vis_id.lower():
        vis_type = "Matrice de Corrélation"
    elif 'distribution' in vis_id.lower() or 'hist' in vis_id.lower():
        vis_type = "Graphique de Distribution"
    elif 'scatter' in vis_id.lower() or 'relation' in vis_id.lower():
        vis_type = "Nuage de Points"
    elif 'box' in vis_id.lower():
        vis_type = "Boîte à Moustaches"

    # Récupérer le titre
    title = vis.get("title", vis_type)

    # Extraire la prompt utilisateur

```

```

user_prompt = agent1_data.get("user_prompt", "")

# Extraire des informations supplémentaires pour le contexte
extra_context = ""
if 'metadata' in vis:
    if 'r_squared' in vis['metadata']:
        extra_context += f"\nR-squared: {vis['metadata']['r_squared']}"
    if 'variables' in vis['metadata']:
        extra_context += f"\nVariables principales: {'', '.join(vis['metadata']['variable

# Créer le prompt pour une interprétation concise
prompt = f"### INTERPRÉTATION CONCISE DE VISUALISATION

### Type
{vis_type}

### Titre
{title}

### Question utilisateur initiale
"{user_prompt}"

### Données visualisées
Variables: {'', '.join(agent1_data["metadata"].get("noms_colonnes", [])[:5]))...
{extra_context}

---

Analyse brièvement cette visualisation en 2-3 phrases maximum. Ton interprétation doit:

1. Identifier les tendances ou relations clés visibles
2. Expliquer comment cette visualisation répond à la question de l'utilisateur
3. Éviter toute description technique de la visualisation elle-même
4. Dans le cas d'un resultat de régression, mentionner les variables clés et leur impact

IMPORTANT: Sois extrêmement concis. Concentre-toi uniquement sur ce qui est pertinent pour r
EXCEPTION : Si c'est une table de régression, tu peux interpreter plus longuement les result
"""

try:
    logger.info(f"Appel à Gemini pour interprétation concise de visualisation {vis_id}")
    interpretation = call_llm(
        prompt=prompt,
        model_name=model,
        backend=backend,
        image_base64=vis['base64']
    )

    logger.info(f"Interprétation générée pour: {vis_id}")
    return interpretation
except Exception as e:
    logger.error(f"Erreur lors de l'interprétation de l'image pour {vis_id}: {e}")
    return f"Erreur d'interprétation: {e}"

# =====
# Fonctions principales de génération des rapports
# =====

def generate_report_docx(agent1_data, agent2_data, synthesis, discussion, conclusion, refere
"""
Génère un rapport Word au format simple avec python-docx.

Args:
    agent1_data: Données de l'agent1
    agent2_data: Données de l'agent2

```

```
synthesis: Résumé simple
discussion: Section discussion
conclusion: Section conclusion
references: Références bibliographiques
economic_reasoning: Raisonnement économique complet
model_name: Nom du modèle LLM utilisé
user_prompt: Prompt initial de l'utilisateur
img_dir: Répertoire contenant les images
```

Returns:

```
    str: Chemin du rapport Word généré ou message d'erreur
"""
if not DOCX_AVAILABLE:
    logger.error("Module python-docx non disponible. Impossible de générer le rapport Wo
    return None

timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
report_dir = os.path.join("outputs", f"rapport_{timestamp}")
os.makedirs(report_dir, exist_ok=True)

# Créer un nouveau document Word
doc = Document()

# Définir le style de base
style = doc.styles['Normal']
style.font.name = 'Calibri'
style.font.size = Pt(11)

# Fonction pour convertir le Markdown en texte brut
h = html2text.HTML2Text()
h.ignore_links = True
h.ignore_images = True
h.ignore_emphasis = True
h.body_width = 0 # No wrapping

def markdown_to_plain_text(markdown_text):
    """Convertit le Markdown en texte brut"""
    if not markdown_text:
        return ""
    # Supprimer les # des titres
    text = re.sub(r'^#\s+', '', markdown_text, flags=re.MULTILINE)
    # Supprimer les * et _ d'emphase
    text = re.sub(r'(\*|_)(.*?)\1', r'\2', text)
    text = re.sub(r'(\*|_)(.*?)\1', r'\2', text)
    # Supprimer les ``` des blocs de code
    text = re.sub(r'```\s*\n(.*?)\s```', r'\1', text, flags=re.DOTALL)
    # Simplifier les listes
    text = re.sub(r'^\s*[-*]\s+', '• ', text, flags=re.MULTILINE)
    text = re.sub(r'^\s*\d+\.\s+', '• ', text, flags=re.MULTILINE)

    return text

# Page de titre améliorée
# Ajouter un titre plus élégant
title_paragraph = doc.add_paragraph()
title_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
title_run = title_paragraph.add_run(f"Analyse Économique")
title_run.font.size = Pt(28)
title_run.font.bold = True
title_run.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu

# Ajouter un sous-titre avec le prompt utilisateur
```

```

subtitle_paragraph = doc.add_paragraph()
subtitle_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
subtitle_run = subtitle_paragraph.add_run(f"{user_prompt[:80]}{'...' if len(user_prompt)
subtitle_run.font.size = Pt(16)
subtitle_run.italic = True
subtitle_run.font.color.rgb = RGBColor(0x4B, 0x55, 0x63) # Gris foncé

# Ajouter une ligne horizontale décorative
border_paragraph = doc.add_paragraph()
border_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
border_run = border_paragraph.add_run("■■■■■■■■■■■■■■■■■■■■")
border_run.font.size = Pt(16)
border_run.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair

# Ajouter "Rapport d'analyse économique"
doc_type_paragraph = doc.add_paragraph()
doc_type_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
doc_type_run = doc_type_paragraph.add_run("Rapport d'analyse économique")
doc_type_run.font.size = Pt(14)
doc_type_run.font.color.rgb = RGBColor(0x6B, 0x72, 0x80) # Gris moyen

# Ajouter la date
date_paragraph = doc.add_paragraph()
date_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
date_paragraph.add_run(f"Généré le {datetime.now().strftime('%d/%m/%Y')}")

# Ajouter le modèle utilisé
model_paragraph = doc.add_paragraph()
model_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
model_run = model_paragraph.add_run(f"Analyse réalisée avec {model_name}")
model_run.font.size = Pt(9)
model_run.font.color.rgb = RGBColor(0x6B, 0x72, 0x80) # Gris moyen

doc.add_page_break()

# Résumé
heading = doc.add_heading("Résumé", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu
doc.add_paragraph(markdown_to_plain_text(synthesis))

# Introduction
heading = doc.add_heading("Introduction", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu
doc.add_paragraph(f"Cette analyse s'intéresse à {user_prompt.lower()}")
intro_text = agent1_data.get('llm_output', {}).get('introduction', 'Non disponible')
doc.add_paragraph(markdown_to_plain_text(intro_text))

# Visualisations et Résultats

doc.add_page_break()
heading = doc.add_heading("Visualisations et Résultats", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu
doc.add_paragraph("Les visualisations ci-dessous illustrent les relations entre les diff

# Traiter les visualisations
visualisations_data = agent2_data.get("visualisations", [])
standard_visualizations = [v for v in visualisations_data if v.get('type') != 'regressio
regression_tables = [v for v in visualisations_data if v.get('type') == 'regression_tabl

# Parcourir les visualisations standard
for i, vis in enumerate(standard_visualizations):
    if 'filename' in vis:
        title = vis.get('title', f'Figure {i+1}')

```

```

heading = doc.add_heading(title, level=2)
heading.style.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair

# Ajouter l'image
img_path = os.path.join(img_dir, vis.get('filename'))
if os.path.exists(img_path):
    try:
        doc.add_picture(img_path, width=Inches(6.0))
        last_paragraph = doc.paragraphs[-1]
        last_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
    except Exception as e:
        logger.error(f"Erreur lors de l'ajout de l'image au document Word: {e}")

# Ajouter l'interprétation
if 'interpretation' in vis and vis['interpretation']:
    interp_para = doc.add_paragraph()
    interp_para.add_run("Interprétation: ").bold = True
    interp_para.add_run(markdown_to_plain_text(vis['interpretation']))

# Résultats des régressions
if regression_tables:
    doc.add_page_break()
    heading = doc.add_heading("Résultats des Régressions", level=1)
    heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu

    # Texte d'introduction mettant l'accent sur l'importance des régressions
    intro_para = doc.add_paragraph()
    intro_para.add_run("Les modèles de régression présentés ci-dessous constituent ").italic = True
    emphasis_run = intro_para.add_run("le cœur de notre analyse économétrique")
    emphasis_run.italic = True
    emphasis_run.bold = True
    emphasis_run.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu
    intro_para.add_run(". Ils permettent d'analyser de manière rigoureuse les relations").italic = True

    # Ajouter une ligne horizontale pour marquer l'importance de cette section
    border_paragraph = doc.add_paragraph()
    border_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
    border_run = border_paragraph.add_run("_" * 40)
    border_run.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair

    for i, table in enumerate(regression_tables):
        title = table.get('title', f'Régression {i+1}')
        heading = doc.add_heading(title, level=2)
        heading.style.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair

        # Ajouter l'image de la régression
        filename = table.get('filename')
        if filename is None:
            # Utiliser l'ID comme fallback ou générer un nom par défaut
            filename = f"{table.get('id', f'regression_{i+1}')}.png"
        img_path = os.path.join(img_dir, filename)
        if os.path.exists(img_path):
            try:
                doc.add_picture(img_path, width=Inches(6.0))
                last_paragraph = doc.paragraphs[-1]
                last_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
            except Exception as e:
                logger.error(f"Erreur lors de l'ajout de l'image de régression au document Word: {e}")

        # Ajouter l'interprétation détaillée si disponible
        if 'detailed_interpretation' in table and table['detailed_interpretation']:
            heading_interp = doc.add_heading("Interprétation économétrique détaillée", level=3)
            heading_interp.style.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair

            # Ajout d'un style spécial pour la section d'interprétation

```

```

para_interp = doc.add_paragraph()
para_interp.style = doc.styles['Normal']
para_interp.paragraph_format.left_indent = Inches(0.2)
para_interp.paragraph_format.first_line_indent = Inches(-0.2)

# Ajout d'un symbole pour marquer le début de l'interprétation
para_interp.add_run("■ ").bold = True

# Ajout du contenu de l'interprétation
para_interp.add_run(markdown_to_plain_text(table['detailed_interpretation']))

# Sinon, utiliser l'interprétation standard si disponible
elif 'interpretation' in table and table['interpretation']:
    interp_para = doc.add_paragraph()
    interp_para.add_run("Interprétation: ").bold = True
    interp_para.add_run(markdown_to_plain_text(table['interpretation']))

# Ajouter un tableau pour les coefficients significatifs si disponible
if 'data' in table and table['data'] and 'coefficients' in table['data']:
    coefficients = table['data']['coefficients']
    r_squared = table['data'].get('r_squared', 'N/A')

    # Ajouter une ligne pour le R-squared
    r_squared_para = doc.add_paragraph()
    r_squared_para.add_run("R-squared: ").bold = True
    r_squared_para.add_run(r_squared)

    # Ajouter un tableau pour les coefficients
    doc.add_heading("Coefficients significatifs", level=4)
    coef_table = doc.add_table(rows=1, cols=4)
    coef_table.style = 'Table Grid'

    # En-têtes du tableau
    header_cells = coef_table.rows[0].cells
    header_cells[0].text = "Variable"
    header_cells[1].text = "Coefficient"
    header_cells[2].text = "P-value"
    header_cells[3].text = "Significativité"

    # Données des coefficients
    for coef in coefficients:
        row_cells = coef_table.add_row().cells
        row_cells[0].text = coef.get('variable', 'N/A')
        row_cells[1].text = coef.get('coef', 'N/A')
        p_value = coef.get('p_value', 'N/A')
        row_cells[2].text = p_value

        # Déterminer si le coefficient est significatif
        is_significant = False
        if p_value != 'N/A':
            try:
                is_significant = float(p_value) < 0.05
            except ValueError:
                pass

        row_cells[3].text = "Significatif" if is_significant else "Non significatif"

    # Mettre en évidence les coefficients significatifs
    if is_significant:
        for cell in row_cells:
            for paragraph in cell.paragraphs:
                for run in paragraph.runs:
                    run.bold = True

```

```

run.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu

# Ajouter un séparateur entre les régressions
if i < len(regression_tables) - 1:
    doc.add_paragraph()
    separator = doc.add_paragraph()
    separator.alignment = WD_ALIGN_PARAGRAPH.CENTER
    separator.add_run("* * *")
    doc.add_paragraph()

# Analyse globale
doc.add_page_break()
heading = doc.add_heading("Analyse globale", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu
comprehensive_visual_analysis = generate_comprehensive_visual_analysis(
    visualisations_data,
    agent1_data,
    agent2_data,
    model_name,
    "ollama" if "ollama" in model_name.lower() else "gemini"
)
doc.add_paragraph(markdown_to_plain_text(comprehensive_visual_analysis))

# Raisonnement économique approfondi
doc.add_page_break()
heading = doc.add_heading("Raisonnement économique approfondi", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu

# Convertir le raisonnement économique en sections formatées
eco_reasoning_lines = markdown_to_plain_text(economic_reasoning).split('\n')
current_level = 1
for line in eco_reasoning_lines:
    line = line.strip()
    if not line:
        continue

    # Détecter si c'est un titre
    if line.startswith("Synthèse globale"):
        heading = doc.add_heading("Synthèse globale", level=2)
        heading.style.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair
        current_level = 2
    elif line.startswith("Analyse économique approfondie"):
        heading = doc.add_heading("Analyse économique approfondie", level=2)
        heading.style.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair
        current_level = 2
    elif line.startswith("Limites et nuances"):
        heading = doc.add_heading("Limites et nuances", level=2)
        heading.style.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair
        current_level = 2
    elif line.startswith("Implications pratiques et théoriques"):
        heading = doc.add_heading("Implications pratiques et théoriques", level=2)
        heading.style.font.color.rgb = RGBColor(0x3B, 0x82, 0xF6) # Bleu clair
        current_level = 2
    # Détecter les listes
    elif line.startswith('• '):
        para = doc.add_paragraph(line[2:], style='List Bullet')
    # Sinon c'est un paragraphe normal
    else:

        doc.add_paragraph(line)

# Discussion
doc.add_page_break()

```

```

heading = doc.add_heading("Discussion", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu
doc.add_paragraph("Cette section interprète les résultats de l'analyse et discute leurs
doc.add_paragraph(markdown_to_plain_text(discussion))

# Conclusion
heading = doc.add_heading("Conclusion", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu
doc.add_paragraph(markdown_to_plain_text(conclusion))

# Références
doc.add_page_break()
heading = doc.add_heading("Références", level=1)
heading.style.font.color.rgb = RGBColor(0x25, 0x63, 0xEB) # Bleu

# Traiter les références ligne par ligne
refs_clean = markdown_to_plain_text(references)
for line in refs_clean.split('\n'):
    line = line.strip()
    if line.startswith('• '):
        doc.add_paragraph(line[2:], style='List Bullet')
    elif line:
        doc.add_paragraph(line)

# Pied de page
footer_paragraph = doc.add_paragraph()
footer_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
footer_run = footer_paragraph.add_run(f"Rapport généré avec {model_name} | {datetime.now}
footer_run.font.size = Pt(8)
footer_run.font.color.rgb = RGBColor(0x6B, 0x72, 0x80) # Gris moyen

# Enregistrer le document
docx_path = os.path.join(report_dir, "rapport.docx")
try:
    doc.save(docx_path)
    logger.info(f"Rapport Word généré avec succès: {docx_path}")
    return docx_path
except Exception as e:
    logger.error(f"Erreur lors de la génération du document Word: {e}")
    return f"Erreur: Échec de la génération du document Word. Détails: {e}"

def generate_report_pdf(agent1_data, agent2_data, synthesis, discussion, conclusion, referen
"""
Génère un rapport PDF au format amélioré avec WeasyPrint et Jinja2.

Args:
    agent1_data: Données de l'agent1
    agent2_data: Données de l'agent2
    synthesis: Résumé simple
    discussion: Section discussion
    conclusion: Section conclusion
    references: Références bibliographiques
    economic_reasoning: Raisonnement économique complet
    model_name: Nom du modèle LLM utilisé
    user_prompt: Prompt initial de l'utilisateur
    report_dir: Répertoire où générer le rapport
    img_dir: Répertoire où sauvegarder les images

Returns:
    str: Chemin du rapport PDF généré ou message d'erreur
"""
logger.info(f"Répertoire du rapport: {report_dir}")

# --- Préparation des données pour le template ---
narrative_md = agent2_data.get("narrative", "Aucune narration disponible.")

```

```

# Traiter les visualisations (prend en charge à la fois les visualisations standard et les
visualisations_data = agent2_data.get("visualisations", [])

# Log pour le débogage
logger.info(f"Nombre de visualisations dans agent2_data: {len(visualisations_data)}")
for i, vis in enumerate(visualisations_data):
    logger.info(f"Visualisation {i+1}: type={vis.get('type', 'non spécifié')}, base64={vis.get('base64', '')}")
    logger.info(f"  Données structurées: {'présentes' if 'data' in vis and vis['data'] else 'absentes'}")
    logger.info(f"  Données CSV: {'présentes' if 'csv_data' in vis and vis['csv_data'] else 'absentes'}")
    logger.info(f"  Interprétation: {'présente' if 'interpretation' in vis and vis['interpretation'] else 'absente'}")

# Séparer les visualisations et les tables de régression
standard_visualizations = [v for v in visualisations_data if v.get('type') != 'regression_table']
regression_tables = [v for v in visualisations_data if v.get('type') == 'regression_table']

# Sauvegarder les images et obtenir leurs infos (filename, title)
image_infos = save_images(visualisations_data, img_dir)

logger.info(f"Nombre d'images sauvegardées: {len(image_infos)}")
for i, img in enumerate(image_infos):
    logger.info(f"Image sauvegardée {i+1}: filename={img.get('filename')}, type={img.get('type', 'non spécifié')}")
    logger.info(f"  Données structurées: {'présentes' if 'data' in img and img['data'] else 'absentes'}")
    logger.info(f"  Données CSV: {'présentes' if 'csv_data' in img and img['csv_data'] else 'absentes'}")
    logger.info(f"  Interprétation: {'présente' if 'interpretation' in img and img['interpretation'] else 'absente'}")

# Séparer les informations sur les tables de régression et les visualisations standard
standard_vis_infos = [img for img in image_infos if img.get('type') != 'regression_table']
regression_table_infos = [img for img in image_infos if img.get('type') == 'regression_table']

# Extraire et formater la table de régression à partir du texte narrative

regression_table_html = parse_regression_to_html(narrative_md, regression_table_infos)

# Vérifier si un modèle puissant a été utilisé par l'agent 2
if "current_model" in agent2_data and agent2_data["current_model"] != model_name:
    logger.info(f"Utilisation du modèle puissant transmis par l'agent 2: {agent2_data['current_model']}")
    model_name = agent2_data["current_model"]

# Generate a comprehensive visual analysis if needed
comprehensive_visual_analysis = ""
if len(image_infos) > 1: # Only generate if we have multiple visualizations
    logger.info("Génération d'une analyse visuelle globale")
    # Déterminer le backend basé sur le modèle
    backend = "ollama" if "ollama" in model_name.lower() else "gemini"
    comprehensive_visual_analysis = generate_comprehensive_visual_analysis(
        image_infos,
        agent1_data,
        agent2_data,
        model_name,
        backend
    )

# --- Création du HTML de rapport avec un template amélioré ---
# Nouveau template HTML avec les améliorations demandées
html_template = """<!DOCTYPE html>
<html lang="fr">
<head>
<meta charset="UTF-8">
<title>{{ report_title }}</title>
<style>
/* Style de base simplifié */
body {
font-family: Arial, sans-serif;

```

```

    font-size: 10pt;
    line-height: 1.3;
    color: #333;
    margin: 1.5cm;
}

/* En-tête et pied de page */
@page {
    size: A4;
    margin: 2cm 1.5cm;
    @top-center {
        content: "{ { report_title | replace(' ', '') } }";
        font-family: Arial, sans-serif;
        font-size: 8pt;
        color: #666;
    }
    @bottom-center {
        content: "Page " counter(page) " / " counter(pages);
        font-family: Arial, sans-serif;
        font-size: 8pt;
        color: #666;
    }
}

/* Typographie simple */
h1 {
    font-size: 14pt;
    margin-top: 1em;
    margin-bottom: 0.5em;
    color: #000;
    border-bottom: 1px solid #ccc;
    padding-bottom: 3px;
}

h2 {
    font-size: 12pt;
    margin-top: 0.8em;
    margin-bottom: 0.4em;
    color: #000;
}

h3 {
    font-size: 11pt;
    margin-top: 0.7em;
    margin-bottom: 0.3em;
    color: #000;
}

h4 {
    font-size: 10pt;
    margin-top: 0.6em;
    margin-bottom: 0.3em;
    color: #000;
}

p {
    margin: 0.4em 0 0.8em 0;
    text-align: justify;
}

/* Images */
img {
    max-width: 100%;
    height: auto;
    display: block;
}

```

```

    margin: 0.8em auto;
}

/* Tableaux simplifiés */
table {
    width: 100%;
    border-collapse: collapse;
    margin: 0.8em 0;
    font-size: 9pt;
}

table th {
    background-color: #f2f2f2;
    font-weight: bold;
    text-align: left;
    padding: 4px;
    border: 1px solid #ddd;
}

table td {
    padding: 4px;
    border: 1px solid #ddd;
}

/* Listes */
ul, ol {
    margin: 0.5em 0 0.8em 1.2em;
    padding-left: 0;
}

li {
    margin-bottom: 0.2em;
}

/* Saut de page */
.page-break-before {
    page-break-before: always;
}

/* Pied de page simple */
.footer {
    text-align: center;
    font-size: 8pt;
    color: #666;
    margin-top: 1em;
    padding-top: 5px;
    border-top: 1px solid #ddd;
}

/* Tableaux CSV */
.csv-data-table {
    font-size: 8pt;
}

/* Suppression des styles complexes, ne garder que l'essentiel */
.cover-page {
    margin-bottom: 2cm;
}

.cover-title-main {
    font-size: 18pt;
    font-weight: bold;
    text-align: center;
}

```

```

        margin-bottom: 0.5cm;
    }

    .cover-title-secondary {
        font-size: 14pt;
        text-align: center;
        margin-bottom: 1cm;
    }

    .cover-date {
        font-size: 10pt;
        text-align: center;
        margin-top: 1cm;
    }

    .cover-model {
        font-size: 9pt;
        text-align: center;
        color: #666;
        margin-top: 2cm;
    }

    /* Figure container simplifié */
    .figure-container {
        margin: 1em 0;
    }

    .figure-title {
        font-weight: bold;
        font-size: 10pt;
        text-align: center;
        margin: 0.4em 0;
    }

    .figure-interpretation {
        margin: 0.4em 0;
        font-style: italic;
        font-size: 9pt;
    }
}
</style>

```

```

</head>
<body>
    <!-- Page de couverture simplifiée -->
    <div class="cover-page">
        <div class="cover-title-main">Analyse Économique</div>
        <div class="cover-title-secondary">{{ user_prompt }}</div>
        <div class="cover-date">Généré le {{ generation_date }}</div>
        <div class="cover-model">Analyse réalisée avec {{ model_name }}</div>
    </div>

    <!-- Contenu principal -->
    <!-- Résumé -->
    <h1>Résumé</h1>
    {{ synthesis | markdown | safe }}

    <!-- Introduction -->
    <h1>Introduction</h1>
    <p>Cette analyse s'intéresse à {{ user_prompt | lower }}.</p>
    {{ introduction_text | markdown | safe }}

    <!-- Visualisations -->
    <h1 class="page-break-before">Visualisations et Résultats</h1>

```

```

<p>Les visualisations ci-dessous illustrent les relations entre les différentes variables

{% for vis in standard_vis_infos %}
<h2>{{ vis.title }}</h2>
<div class="figure-container">
  
</div>
{% if vis.interpretation %}
<p class="figure-interpretation">{{ vis.interpretation | markdown | safe }}</p>
{% endif %}
{% endfor %}

<!-- Résultats des régressions -->
{% if regression_table_infos %}
<h1 class="page-break-before">Résultats des Régressions</h1>

{% for table in regression_table_infos %}
<h2>{{ table.title }}</h2>
<div class="figure-container">
  
</div>

{% if table.detailed_interpretation %}
<h3>Interprétation économétrique détaillée</h3>
{{ table.detailed_interpretation | markdown | safe }}
{% elif table.interpretation %}
<p class="figure-interpretation">{{ table.interpretation | markdown | safe }}</p>
{% endif %}

{% if table.data and table.data.coefficients %}
<h3>Coefficients significatifs</h3>
<table>
  <thead>
    <tr>
      <th>Variable</th>
      <th>Coefficient</th>
      <th>p-value</th>
      <th>Significativité</th>
    </tr>
  </thead>
  <tbody>
    {% for coef in table.data.coefficients %}
    {% set is_significant = coef.p_value|float < 0.05 if coef.p_value != 'N/A' else False %}
    <tr>
      <td>{{ coef.variable }}</td>
      <td>{{ coef.coef }}</td>
      <td>{{ coef.p_value }}</td>
      <td>{{ "Significatif" if is_significant else "Non significatif" }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
{% endif %}
{% endfor %}
{% endif %}

<!-- Analyse globale -->
{% if comprehensive_visual_analysis %}
<h1 class="page-break-before">Analyse globale</h1>
{{ comprehensive_visual_analysis | markdown | safe }}
{% endif %}

<!-- Raisonnement économique -->
<h1 class="page-break-before">Raisonnement économique</h1>
{{ economic_reasoning | markdown | safe }}

```

```

<!-- Discussion -->
<h1 class="page-break-before">Discussion</h1>
{{ discussion_text | markdown | safe }}

<!-- Conclusion -->
<h1>Conclusion</h1>
{{ conclusion_text | markdown | safe }}

<!-- Références -->
<h1 class="page-break-before">Références</h1>
{{ references | markdown | safe }}


<!-- Footer -->
<div class="footer">
    Rapport généré avec {{ model_name }} | {{ generation_date }}
</div>
</body>
</html>""

# --- Configuration de Jinja2 ---
# Créer un environnement Jinja2 à partir d'une chaîne
env = Environment(loader=FileSystemLoader('.'))
env.filters['basename'] = basename_filter
env.filters['markdown'] = markdown_filter

# Créer un template à partir de la chaîne HTML
template = env.from_string(html_template)

# --- Données à passer au template ---
template_data = {
    "report_title": f"Analyse de {user_prompt[:50]}{'...' if len(user_prompt) > 50 else ''}",
    "generation_date": datetime.now().strftime("%d/%m/%Y"),
    "metadata": agent1_data.get("metadata", {}),
    "introduction_text": agent1_data.get('llm_output', {}).get('introduction', 'Non disponible'),
    "standard_vis_infos": standard_vis_infos,
    "regression_table_infos": regression_table_infos,
    "comprehensive_visual_analysis": comprehensive_visual_analysis,
    "economic_reasoning": economic_reasoning,
    "synthesis": synthesis,
    "discussion_text": discussion,
    "conclusion_text": conclusion,
    "references": references,
    "model_name": model_name,
    "user_prompt": user_prompt
}

# --- Génération du HTML final ---
try:
    final_html = template.render(template_data)
    html_path = os.path.join(report_dir, "rapport.html")
    with open(html_path, "w", encoding="utf-8") as f:
        f.write(final_html)
    logger.info(f"HTML généré: {html_path}")

except Exception as e:
    logger.error(f"Erreur lors du rendu du template: {e}")
    return f"Erreur: Le rendu du template a échoué. Détails: {e}"

# --- Génération du PDF avec les styles améliorés ---
pdf_path = os.path.join(report_dir, "rapport.pdf")
try:
    # Style CSS pour le PDF
    enhanced_css = CSS(string="")

```

```

@page {
    size: A4;
    margin: 2cm 1.5cm;
    @top-center {
        content: string(heading);
        font-family: Arial, sans-serif;
        font-size: 8pt;
        color: #666;
    }
    @bottom-center {
        content: "Page " counter(page) " / " counter(pages);
        font-family: Arial, sans-serif;
        font-size: 8pt;
        color: #666;
    }
}

h1 { string-set: heading content() }

/* Optimisations de base */
body {
    font-family: Arial, sans-serif;
    font-size: 10pt;
    line-height: 1.3;
    margin: 0;
    color: #333;
}

p {
    margin-bottom: 0.5em;
    text-align: justify;
}

h1, h2, h3, h4 {
    page-break-after: avoid;
    margin-bottom: 0.5em;
}

/* Éviter les sauts de page entre titres et contenu */
h1 + p, h2 + p, h3 + p {
    page-break-before: avoid;
}

/* Simplicité pour les figures */
.figure-container {
    margin: 1em 0;
    page-break-inside: avoid;
}

/* Forcer les sauts de page uniquement quand nécessaire */
.page-break-before {
    page-break-before: always;
}

/* Optimisations pour les tableaux */
table {
    margin-bottom: 0.8em;
    page-break-inside: avoid;
}

/* Meilleure gestion des listes */
ul, ol {
    margin-bottom: 0.6em;
}

```

```

        }""")

    # Options pour WeasyPrint
    html_obj = HTML(string=final_html, base_url=report_dir)
    html_obj.write_pdf(pdf_path, stylesheets=[enhanced_css])

    logger.info(f"Rapport PDF généré avec succès: {pdf_path}")
    return pdf_path
except Exception as e:
    logger.error(f"Erreur lors de la génération du PDF avec WeasyPrint: {e}")
    try:
        # Tentative simplifiée sans le CSS amélioré
        HTML(string=final_html, base_url=report_dir).write_pdf(pdf_path)
        logger.info(f"Rapport PDF généré (mode dégradé): {pdf_path}")
        return pdf_path
    except Exception as e2:
        logger.error(f"Échec de la génération PDF même en mode dégradé: {e2}")
        return f"Erreur: Échec de la génération PDF. Détails: {e} / {e2}"

def main():
    """
    Fonction principale qui orchestre la génération du rapport.
    """
    parser = argparse.ArgumentParser(description="Agent 3: Synthèse et Rapport PDF")
    parser.add_argument("agent1_output", help="Fichier JSON généré par l'agent 1")
    parser.add_argument("agent2_output", help="Fichier JSON généré par l'agent 2")
    parser.add_argument("user_prompt", help="Prompt utilisateur original")
    parser.add_argument("--model", default="gemma3:27b", help="Modèle LLM à utiliser")
    parser.add_argument("--backend", default="ollama", choices=["ollama", "gemini"], help="Backend")
    parser.add_argument("--log-file", help="Fichier de log spécifique") # Nouvel argument
    args = parser.parse_args()

    # Reconfigurer le logging si un fichier de log spécifique est fourni
    if args.log_file:
        # Supprimer les handlers existants
        for handler in logger.handlers[:]:
            logger.removeHandler(handler)

        # Ajouter les nouveaux handlers
        file_handler = logging.FileHandler(args.log_file, mode='a') # mode 'a' pour append
        file_handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s'))
        logger.addHandler(file_handler)

        stream_handler = logging.StreamHandler(sys.stderr)
        stream_handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s'))
        logger.addHandler(stream_handler)

        logger.info(f"Logging redirigé vers {args.log_file}")

    # Lecture des données des agents précédents
    try:
        with open(args.agent1_output, "r", encoding="utf-8") as f:
            agent1_data = json.load(f)
    except Exception as e:
        logger.error(f"Erreur lors de la lecture du fichier de l'agent 1: {e}")
        sys.exit(1)

    try:
        with open(args.agent2_output, "r", encoding="utf-8") as f:
            agent2_data = json.load(f)
    except Exception as e:
        logger.error(f"Erreur lors de la lecture du fichier de l'agent 2: {e}")
        sys.exit(1)

    # Vérifier si un modèle puissant a été utilisé par l'agent 2

```

```

model = args.model
if "current_model" in agent2_data and agent2_data["current_model"] != args.model:
    logger.info(f"Utilisation du modèle puissant transmis par l'agent 2: {agent2_data['current_model']}")
    model = agent2_data["current_model"]

# Générer le contenu
logger.info(f"Génération du résumé avec modèle: {model}")
synthesis = generate_synthesis(agent1_data, agent2_data, model, args.backend)

logger.info(f"Génération de la section discussion avec modèle: {model}")
discussion = generate_discussion_section(agent1_data, agent2_data, model, args.backend)

logger.info(f"Génération de la conclusion avec modèle: {model}")
conclusion = generate_conclusion_section(agent1_data, agent2_data, model, args.backend)

logger.info(f"Génération des références avec modèle: {model}")
references = generate_references(agent1_data, model, args.backend)

# AJOUT: Génération du raisonnement économique complet
logger.info(f"Génération du raisonnement économique complet avec modèle: {model}")
economic_reasoning = generate_comprehensive_economic_analysis(agent1_data, agent2_data, model, args.backend)

# Création des répertoires pour les rapports
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
report_dir = os.path.join("outputs", f"rapport_{timestamp}")
img_dir = os.path.join(report_dir, "images")
os.makedirs(report_dir, exist_ok=True)
os.makedirs(img_dir, exist_ok=True)
logger.info(f"Répertoire du rapport: {report_dir}")

# Mise à jour des interprétations des visualisations pour qu'elles soient plus concises
logger.info("Mise à jour des interprétations de visualisations pour plus de concision")

# Mise à jour des interprétations dans les visualisations
visualisations_data = agent2_data.get("visualisations", [])
prompts_log_path = "outputs/prompts_interpretations.txt"

# Dossier temporaire pour sauvegarder les images
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
temp_img_dir = os.path.join("outputs", f"temp_images_{timestamp}")
os.makedirs(temp_img_dir, exist_ok=True)

# Sauvegarder temporairement les images pour pouvoir les interpréter
updated_visualisations = []

for vis in visualisations_data:
    if 'base64' in vis:
        try:
            # Générer une interprétation plus concise
            interpretation = interpret_visualization_with_gemini(
                vis,
                agent1_data,
                model,
                args.backend,
                prompts_log_path
            )

            # Mise à jour de l'interprétation
            vis['interpretation'] = interpretation
            updated_visualisations.append(vis)
            logger.info(f"Interprétation mise à jour pour {vis.get('id', vis.get('filename', ''))}")
        except Exception as e:
            logger.error(f"Erreur lors de la mise à jour de l'interprétation: {e}")

```

```

        updated_visualisations.append(vis) # Garder la visualisation même en cas d'
    else:
        logger.warning(f"Pas de données base64 pour {vis.get('id', vis.get('filename', '')}")
        updated_visualisations.append(vis)

# Mettre à jour les visualisations dans agent2_data
agent2_data["visualisations"] = updated_visualisations

logger.info("Génération du rapport PDF")
pdf_path = generate_report_pdf(
    agent1_data,
    agent2_data,
    synthesis,
    discussion,
    conclusion,
    references,
    economic_reasoning,
    model,
    args.user_prompt,
    report_dir,
    img_dir
)

# Génération du rapport Word si le module est disponible
docx_path = None
if DOCX_AVAILABLE:
    logger.info("Génération du rapport Word")
    docx_path = generate_report_docx(
        agent1_data,
        agent2_data,
        synthesis,
        discussion,
        conclusion,
        references,
        economic_reasoning,
        model,
        args.user_prompt,
        img_dir
    )
else:
    logger.warning("Module python-docx non disponible. Aucun rapport Word ne sera généré")

# Sortie JSON contenant les chemins et la synthèse
output = {
    "abstract": synthesis,
    "discussion": discussion,
    "conclusion": conclusion,
    "references": references,
    "economic_reasoning": economic_reasoning,
    "rapport_pdf": pdf_path if isinstance(pdf_path, str) and pdf_path.endswith(".pdf") else None,
    "rapport_docx": docx_path if isinstance(docx_path, str) and docx_path.endswith(".docx") else None,
    "error": pdf_path if not (isinstance(pdf_path, str) and pdf_path.endswith(".pdf")) else None,
    "model_used": model
}

print(json.dumps(output, ensure_ascii=False, indent=2))

if output["error"]:

    logger.error(f"Le pipeline s'est terminé mais la génération du PDF a échoué: {output['error']}")
    sys.exit(1)

if not DOCX_AVAILABLE:
```

```
        logger.warning("La génération du document Word est désactivée car python-docx n'est  
elif not output["rapport_docx"]:  
        logger.warning("La génération du document Word a échoué")  
  
if __name__ == "__main__":  
    main()
```

## 7.5 Code Source: llm\_utils.py

Description:

Utilitaires pour l'appel aux modèles de langage (LLM). Supporte à la fois les backends Ollama (local) et Gemini (API cloud). Améliorations: - Robustesse accrue de l'appel API REST Gemini pour les images. - Logging détaillé des requêtes/réponses Gemini. - Option de débogage via variable d'environnement pour forcer l'API REST. - Gestion des réponses bloquées par Gemini. - Timeouts configurables.

Structure du fichier:

Fonctions:

- call\_llm: Appelle un modèle LLM selon le backend spécifié (ollama ou gemini)
- call\_ollama: Appelle un modèle local via la commande Ollama CLI
- call\_gemini: Appelle l'API Gemini de Google, gérant les images via la bibliothèque ou l'API REST
- call\_gemini\_with\_image\_via\_library: Utilise la bibliothèque officielle google
- call\_gemini\_with\_image\_via\_rest: Utilise l'API REST de Gemini pour envoyer une requête avec image
- extract\_text\_from\_gemini\_response: Extrait le texte de manière robuste d'une réponse JSON Gemini réussie

Code source complet:

```
#!/usr/bin/env python3
"""
Utilitaires pour l'appel aux modèles de langage (LLM).
Supporte à la fois les backends Ollama (local) et Gemini (API cloud).

Améliorations:
- Robustesse accrue de l'appel API REST Gemini pour les images.
- Logging détaillé des requêtes/réponses Gemini.
- Option de débogage via variable d'environnement pour forcer l'API REST.
- Gestion des réponses bloquées par Gemini.
- Timeouts configurables.
"""

import subprocess
import os
import requests
import base64
import json
import logging
import time

# Configuration du logging
# Use specific format including module name and line number for better debugging
log_format = '%(asctime)s - %(name)s:%(lineno)d - %(levelname)s - %(message)s'
logging.basicConfig(level=logging.INFO, format=log_format)
logger = logging.getLogger("llm_utils")

# --- Configuration ---
# ■■ Clé API intégrée directement ici (■■ usage local uniquement, ne jamais versionner en p
# Il est FORTEMENT recommandé de charger la clé depuis une variable d'environnement
# ou un système de gestion de secrets en production.
```

```

_DEFAULT_GEMINI_API_KEY = "AIzaSyAYT-NjrJiRK9Ei8gp716uR57CO59puWhg" # Replace with your actu

# Injecte la clé dans l'environnement si elle n'est pas déjà définie
if "GEMINI_API_KEY" not in os.environ:
    os.environ["GEMINI_API_KEY"] = _DEFAULT_GEMINI_API_KEY
    logger.warning("Utilisation de la clé API Gemini par défaut intégrée au code. À utiliser
if "GOOGLE_API_KEY" not in os.environ:
    os.environ["GOOGLE_API_KEY"] = os.environ["GEMINI_API_KEY"] # Pour compatibilité avec l

# --- Constantes ---
DEFAULT_OLLAMA_MODEL = "qwq:32b"
DEFAULT_GEMINI_MODEL = "gemini-1.5-flash-latest" # Utilisation du modèle Flash plus récent
DEFAULT_GEMINI_TIMEOUT = 90 # Secondes (augmenté pour analyse d'image)

# --- Variables de contrôle (pour le débogage) ---
# Mettre à True via variable d'environnement pour forcer l'API REST pour les images
FORCE_GEMINI_REST_FOR_IMAGES = True
# FORCE_GEMINI_REST_FOR_IMAGES = os.getenv('LLM_UTILS_FORCE_GEMINI_REST', 'false').lower() =
if FORCE_GEMINI_REST_FOR_IMAGES:
    logger.warning("LLM_UTILS_FORCE_GEMINI_REST=true: Forçage de l'utilisation de l'API RES

# --- Fonctions principales ---

def call_llm(prompt, model_name=None, backend="ollama", image_base64=None, timeout=None):
    """
    Appelle un modèle LLM selon le backend spécifié (ollama ou gemini).
    Prend en charge l'envoi d'images pour Gemini.

    Args:
        prompt (str): Texte du prompt à envoyer.
        model_name (str, optional): Nom du modèle à utiliser. Défaut selon le backend.
        backend (str): "ollama" ou "gemini". Défaut: "ollama".
        image_base64 (str, optional): Données image encodées en base64 (uniquement pour gemi
        timeout (int, optional): Timeout en secondes pour les appels API Gemini. Défaut: DEF

    Returns:
        str: Réponse du modèle.

    Raises:
        ValueError: Si le backend n'est pas reconnu ou si la clé API est manquante pour Gemi
        RuntimeError: Si l'appel au backend échoue (Ollama ou Gemini).
        ImportError: Si la bibliothèque google.generativeai est nécessaire mais non installé
    """
    logger.debug(f"Appel call_llm - backend: {backend}, modèle: {model_name or 'défaut'}, im

    if backend == "ollama":
        if image_base64:
            logger.warning("Le backend Ollama ne prend pas en charge les images. L'image ser
            model = model_name or DEFAULT_OLLAMA_MODEL
            return call_ollama(model, prompt)

        elif backend == "gemini":
            model = model_name or DEFAULT_GEMINI_MODEL
            api_key = os.getenv("GEMINI_API_KEY")
            if not api_key:
                raise ValueError("[llm_utils] Clé API Gemini (GEMINI_API_KEY) manquante dans l'e

            effective_timeout = timeout if timeout is not None else DEFAULT_GEMINI_TIMEOUT
            return call_gemini(model, prompt, api_key, image_base64, effective_timeout)

        else:
            raise ValueError(f"[llm_utils] Backend non reconnu: '{backend}' (attendu: 'ollama' o

def call_ollama(model, prompt):
    """

```

Appelle un modèle local via la commande Ollama CLI.

Args:

model (str): Nom du modèle Ollama.

prompt (str): Texte du prompt.

Returns:

str: Réponse du modèle (stdout).

Raises:

RuntimeError: Si la commande ollama échoue.

"""

command = ["ollama", "run", model, prompt]

logger.info(f"Exécution Ollama: {' '.join(command)}")

try:

start\_time = time.time()

result = subprocess.run(

command,

capture\_output=True,

text=True,

check=True,

encoding='utf-8' # Assurer l'encodage correct

)

duration = time.time() - start\_time

logger.info(f"Appel Ollama réussi ({duration:.2f}s).")

logger.debug(f"Ollama stdout:\n{result.stdout[:500]}{'...' if len(result.stdout)>500

return result.stdout

except FileNotFoundError:

logger.error("Commande 'ollama' non trouvée. Assurez-vous qu'Ollama est installé et

raise RuntimeError("[llm\_utils] Commande 'ollama' non trouvée.")

except subprocess.CalledProcessError as e:

duration = time.time() - start\_time

logger.error(f"Erreur lors de l'appel Ollama ({duration:.2f}s). stderr:\n{e.stderr.s

raise RuntimeError(f"[llm\_utils] Erreur Ollama (code {e.returncode}): {e.stderr.stri

except Exception as e:

duration = time.time() - start\_time

logger.error(f"Erreur inattendue lors de l'appel Ollama ({duration:.2f}s): {e}", exc

raise RuntimeError(f"[llm\_utils] Erreur inattendue Ollama: {e}")

def call\_gemini(model, prompt, api\_key, image\_base64=None, timeout=DEFAULT\_GEMINI\_TIMEOUT):

"""

Appelle l'API Gemini de Google, gérant les images via la bibliothèque ou l'API REST.

Args:

model (str): Nom du modèle Gemini (ex: "gemini-1.5-flash-latest").

prompt (str): Texte du prompt.

api\_key (str): Clé API Google.

image\_base64 (str, optional): Image encodée en base64.

timeout (int): Timeout en secondes pour l'appel API.

Returns:

str: Réponse textuelle du modèle.

Raises:

RuntimeError: Si l'appel API échoue ou la réponse est invalide.

ImportError: Si la bibliothèque google.generativeai est requise mais non trouvée.

"""

if image\_base64:

# Décider quelle méthode utiliser pour l'appel avec image

use\_rest\_api = FORCE\_GEMINI\_REST\_FOR\_IMAGES

```

if not use_rest_api:
    try:
        # Tenter d'utiliser la bibliothèque officielle
        logger.info(f"Tentative d'appel Gemini avec image via la bibliothèque google")
        return call_gemini_with_image_via_library(model, prompt, image_base64, api_key, timeout)
    except ImportError:
        logger.warning("Bibliothèque google.generativeai non disponible ou import échoué")
        use_rest_api = True
    except Exception as lib_err:
        # Si la bibliothèque est installée mais échoue pour une autre raison,
        # on peut choisir de passer à REST ou de lever l'erreur.
        # Ici, on passe à REST pour maximiser les chances de succès.
        logger.error(f"Erreur inattendue avec la bibliothèque google.generativeai: {lib_err}")
        use_rest_api = True

if use_rest_api:
    # Utiliser l'API REST comme fallback ou si forcé
    logger.info(f"Appel Gemini avec image via l'API REST (modèle: {model})")
    return call_gemini_with_image_via_rest(model, prompt, image_base64, api_key, timeout)

else:
    # Appel API REST standard sans image
    logger.info(f"Appel Gemini standard (texte seul) via l'API REST (modèle: {model})")
    url = f"https://generativelanguage.googleapis.com/v1beta/models/{model}:generateContent"
    headers = {"Content-Type": "application/json"}
    payload = {"contents": [{"parts": [{"text": prompt}]}]} # Structure simple pour texte seul

    try:
        start_time = time.time()
        response = requests.post(url, headers=headers, json=payload, timeout=timeout)
        duration = time.time() - start_time
        logger.info(f"Appel API Gemini (texte seul) terminé - Statut: {response.status_code} - Durée: {duration:.2f}s")

        response.raise_for_status() # Lève une exception pour les erreurs HTTP 4xx/5xx
        data = response.json()
        logger.debug(f"Réponse JSON Gemini (texte seul):\n{json.dumps(data, indent=2)}")

        # Vérifier les erreurs applicatives dans la réponse JSON
        if 'error' in data:
            logger.error(f"Erreur API Gemini dans la réponse JSON: {json.dumps(data['error'])}")
            raise RuntimeError(f"[llm_utils] Erreur API Gemini: {data['error'].get('message')}")

        # Extraction robuste du texte
        text = extract_text_from_gemini_response(data)
        return text

    except requests.exceptions.Timeout:
        duration = time.time() - start_time
        logger.error(f"Timeout dépassé ({timeout}s) pour l'appel API Gemini (texte seul)")
        raise RuntimeError(f"[llm_utils] Timeout API Gemini ({timeout}s) dépassé.")
    except requests.exceptions.RequestException as e:
        duration = time.time() - start_time if 'start_time' in locals() else 0
        logger.error(f"Erreur de requête API Gemini (texte seul) ({duration:.2f}s): {e}")
        raise RuntimeError(f"[llm_utils] Erreur de requête API Gemini: {e}")
    except Exception as e:
        duration = time.time() - start_time if 'start_time' in locals() else 0
        logger.error(f"Erreur inattendue lors de l'appel Gemini (texte seul) ({duration:.2f}s): {e}")
        raise RuntimeError(f"[llm_utils] Erreur inattendue Gemini: {e}")

def call_gemini_with_image_via_library(model, prompt, image_base64, api_key, timeout):
    """
    Utilise la bibliothèque officielle google.generativeai pour envoyer une requête avec image
    """

```

```

NOTE: Cette fonction lève ImportError si la bibliothèque n'est pas trouvée.
      Elle peut aussi lever d'autres exceptions de la bibliothèque elle-même.
"""
try:
    import google.generativeai as genai
    # from PIL import Image # PIL peut être nécessaire pour certains formats/validations
    import io
except ImportError as e:
    logger.error(f"Dépendance manquante: {e}. Installez google-generativeai.")
    raise ImportError(f"Module google.generativeai non disponible. Installez-le (pip ins

genai.configure(api_key=api_key)

try:
    # Décoder l'image base64 en bytes
    image_bytes = base64.b64decode(image_base64)
    logger.debug(f"Image base64 décodée en {len(image_bytes)} octets.")

    # Préparer l'objet image pour la bibliothèque
    # Utiliser 'blob' semble être la méthode recommandée pour les bytes
    img_blob = {'mime_type': 'image/png', 'data': image_bytes}

    # Créer le contenu multimodal
    contents = [prompt, img_blob]
    logger.debug(f"Contenu envoyé à la bibliothèque Gemini: [prompt, {img_blob['mime_typ

    # Créer le modèle Gemini
    # Ajout de configuration de génération si nécessaire (température, etc.)
    generation_config = genai.types.GenerationConfig(
        # candidate_count=1, # Généralement 1 par défaut
        # stop_sequences=['...'],
        # max_output_tokens=2048,
        temperature=0.5, # Ajuster si besoin
        # top_p=1.0,
        # top_k=1
    )

    # Note: La bibliothèque ne semble pas avoir de timeout direct sur generate_content
    # Le timeout global de la requête HTTP sous-jacente pourrait s'appliquer.
    model_gemini = genai.GenerativeModel(model)

    logger.info("Appel à model.generate_content() via la bibliothèque...")
    start_time = time.time()
    response = model_gemini.generate_content(contents, generation_config=generation_conf
    duration = time.time() - start_time
    logger.info(f"Appel bibliothèque Gemini terminé ({duration:.2f}s).")

    # Log de la réponse brute pour débogage
    try:
        # Tenter d'accéder à des attributs courants pour le log
        response_summary = f"Response(text_len={len(response.text)} if hasattr(response,
        logger.debug(f"Réponse brute de la bibliothèque Gemini: {response_summary}")
        # logger.debug(f"Full response object: {response}") # Attention, peut être très
    except Exception as log_err:
        logger.warning(f"Impossible de générer le résumé de la réponse brute: {log_err}")

    # Vérifier si la réponse a été bloquée
    if hasattr(response, 'prompt_feedback') and response.prompt_feedback.block_reason:
        block_reason = response.prompt_feedback.block_reason
        safety_ratings = getattr(response, 'candidates', [{}])[0].get('safety_ratings',
        logger.error(f"Réponse Gemini bloquée par la bibliothèque. Raison: {block_reason
        raise RuntimeError(f"[llm_utils] Réponse Gemini bloquée (lib): {block_reason}")

    # Extraire et retourner le texte
    if hasattr(response, 'text') and response.text:

```

```

        logger.info("Texte extrait avec succès de la réponse de la bibliothèque.")
        return response.text
    else:
        # Gérer le cas où .text est absent ou vide, peut arriver si bloqué ou autre erreur
        logger.error(f"Aucun attribut 'text' trouvé ou vide dans la réponse de la bibliothèque")
        raise RuntimeError("[llm_utils] Format de réponse inattendu de la bibliothèque Gemini")

except ImportError: # Re-lever pour que l'appelant sache qu'il faut utiliser REST
    raise
except Exception as e:
    # Capturer d'autres erreurs potentielles de la bibliothèque (ex: APIError, InvalidArgumentError)
    logger.error(f"Erreur lors de l'appel à la bibliothèque Gemini: {type(e).__name__}: {e}")
    # Renvoyer l'erreur pour éventuellement tenter avec REST API
    raise RuntimeError(f"[llm_utils] Erreur avec google.generativeai: {e}") from e

def call_gemini_with_image_via_rest(model, prompt, image_base64, api_key, timeout):
    """
    Utilise l'API REST de Gemini pour envoyer une requête avec image.
    Inclut un logging et une gestion d'erreurs robustes.
    """
    url = f"https://generativelanguage.googleapis.com/v1beta/models/{model}:generateContent?"
    headers = {"Content-Type": "application/json"}

    # Création du payload avec texte et image
    payload = {
        "contents": [{
            "parts": [
                {"text": prompt},
                {"inline_data": {"mime_type": "image/png", "data": image_base64}}
            ]
        }],
        # Ajouter generationConfig si nécessaire (température, max tokens etc.)
        "generationConfig": {
            "temperature": 0.5,
            "maxOutputTokens": 4096 # Augmenter pour les descriptions d'images potentiellement longues
        },
        # "topP": 0.8,
        # "topK": 10
    }

    logger.debug(f"Payload JSON envoyé à l'API REST Gemini (image):\n{json.dumps(payload, indent=2)}")

    try:
        start_time = time.time()
        response = requests.post(url, headers=headers, json=payload, timeout=timeout)
        duration = time.time() - start_time

        # Log systématique de la réponse pour débogage
        logger.info(f"Appel API REST Gemini (image) terminé - Statut: {response.status_code}")
        logger.debug(f"Réponse Headers: {response.headers}")
        try:
            # Tenter de décoder en JSON pour le log, même si erreur HTTP
            data = response.json()
            logger.debug(f"Réponse JSON Body:\n{json.dumps(data, indent=2)}")
        except json.JSONDecodeError:
            logger.error(f"La réponse API n'est pas au format JSON. Contenu brut:\n{response.text}")
            # Si ce n'est pas JSON, et que le statut n'est pas 2xx, raise_for_status lèvera l'exception
            # Si le statut est 2xx mais que ce n'est pas JSON, c'est une erreur inattendue.
            if 200 <= response.status_code < 300:
                raise RuntimeError("[llm_utils] Réponse API Gemini (REST) inattendue: Statut 2xx mais pas JSON")

        # Vérifier les erreurs HTTP (4xx, 5xx) après avoir loggé la réponse
        response.raise_for_status()
    except requests.exceptions.Timeout:
        logger.error(f"Délai d'attente dépassé lors de l'appel à l'API REST Gemini (image).")
        raise TimeoutError(f"Délai d'attente dépassé lors de l'appel à l'API REST Gemini (image).")
    except requests.exceptions.HTTPError:
        logger.error(f"Erreur HTTP lors de l'appel à l'API REST Gemini (image). Statut: {response.status_code}")
        raise HTTPError(f"Erreur HTTP lors de l'appel à l'API REST Gemini (image). Statut: {response.status_code}")
    except Exception as e:
        logger.error(f"Erreur inattendue lors de l'appel à l'API REST Gemini (image): {type(e).__name__}: {e}")
        raise RuntimeError(f"Erreur inattendue lors de l'appel à l'API REST Gemini (image): {type(e).__name__}: {e}") from e

```

```

# Si on arrive ici, status code est 2xx et la réponse est JSON
# Vérifier les erreurs applicatives dans la réponse JSON
if 'error' in data:
    logger.error(f"Erreur API Gemini détectée dans la réponse JSON: {json.dumps(data)}")
    raise RuntimeError(f"[llm_utils] Erreur API Gemini (REST): {data['error'].get('message')}")

# Extraction robuste du texte de la réponse
text = extract_text_from_gemini_response(data)
return text

except requests.exceptions.Timeout:
    duration = time.time() - start_time
    logger.error(f"Timeout dépassé ({timeout}s) pour l'appel API REST Gemini (image).")
    raise RuntimeError(f"[llm_utils] Timeout API Gemini ({timeout}s) dépassé.")
except requests.exceptions.RequestException as e:
    duration = time.time() - start_time if 'start_time' in locals() else 0
    logger.error(f"Erreur de requête API REST Gemini (image) ({duration:.2f}s): {e}", exc_info=True)
    raise RuntimeError(f"[llm_utils] Erreur de requête API Gemini (REST): {e}")
except Exception as e: # Capture toute autre erreur (JSONDecodeError si status 2xx, RuntimeError si autre)
    duration = time.time() - start_time if 'start_time' in locals() else 0
    logger.error(f"Erreur inattendue lors de l'appel API REST Gemini (image) ({duration:.2f}s): {e}", exc_info=True)
    # Si ce n'est pas déjà une RuntimeError, on l'enveloppe
    if isinstance(e, RuntimeError):
        raise e
    else:
        raise RuntimeError(f"[llm_utils] Erreur inattendue Gemini (REST): {e}")

def extract_text_from_gemini_response(data):
    """
    Extrait le texte de manière robuste d'une réponse JSON Gemini réussie.

    Args:
        data (dict): Le dictionnaire JSON de la réponse Gemini.

    Returns:
        str: Le texte extrait.

    Raises:
        RuntimeError: Si la structure de la réponse est invalide ou si le contenu est bloqué.
    """
    try:
        if not data or 'candidates' not in data or not isinstance(data['candidates'], list):
            logger.error(f"Structure de réponse Gemini invalide: 'candidates' manquante, non list")
            raise RuntimeError(f"[llm_utils] Format de réponse Gemini invalide: 'candidates'")

        candidate = data['candidates'][0]

        # Vérifier si la génération s'est terminée correctement ou a été bloquée
        finish_reason = candidate.get('finishReason')

        if finish_reason and finish_reason != 'STOP':
            safety_ratings = candidate.get('safetyRatings', [])
            logger.warning(f"Réponse Gemini potentiellement incomplète ou bloquée. FinishReason: {finish_reason}")
            # Selon le cas d'usage, on peut vouloir lever une erreur ou retourner un message
            # Ici, on lève une erreur pour signaler clairement le problème.
            # On pourrait aussi vérifier si 'content' existe malgré le finishReason.
            raise RuntimeError(f"[llm_utils] Réponse Gemini bloquée ou incomplète ({finish_reason})")
            # Alternative : return f"[Contenu bloqué ou incomplet: {finish_reason}]"

        if 'content' not in candidate or 'parts' not in candidate['content'] or not isinstance(candidate['parts'], list):
            logger.error(f"Structure de réponse Gemini invalide: 'content' ou 'parts' manquants")
            raise RuntimeError(f"[llm_utils] Format de réponse Gemini invalide: 'content' ou 'parts'")

        # Extraire le texte des parties
        text_parts = []
        for part in candidate['parts']:
            if 'text' in part:
                text_parts.append(part['text'])

        return '\n'.join(text_parts)
    except Exception as e:
        logger.error(f"Erreur lors de l'extraction du texte: {e}")
        raise

```

```

        if len(candidate['content']['parts']) == 0:
            logger.warning(f"La section 'parts' de la réponse Gemini est vide. Candidat: {js}")
            return "" # Retourner une chaîne vide si parts est vide mais la réponse est valide

        # Extrait le texte de la première partie (le format habituel)
        first_part = candidate['content']['parts'][0]
        if 'text' not in first_part:
            logger.warning(f"Aucune clé 'text' trouvée dans la première partie de la réponse")
            # Peut-être que la réponse est dans une autre partie ? Ou pas de texte du tout.
            # Pour l'instant, on retourne une chaîne vide.
            return ""

        text = first_part['text']
        logger.info(f"Texte extrait avec succès de la réponse Gemini (longueur: {len(text)})")
        return text

    except (KeyError, IndexError, TypeError) as e:
        logger.error(f"Erreur lors de l'analyse de la structure de la réponse Gemini: {e}.R")
        raise RuntimeError(f"[llm_utils] Format de réponse Gemini inattendu lors de l'extraction")

# --- Test ---
if __name__ == "__main__":
    logger.info("--- Début des tests llm_utils ---")

    # === Test 1: Ollama simple ===
    print("\n--- Test 1: Appel Ollama simple ---")
    try:
        # Définir explicitement le modèle pour le test si nécessaire
        test_ollama_model = DEFAULT_OLLAMA_MODEL # Ou un modèle plus petit comme "phi3" ou "phi3.5"
        response_ollama = call_llm("Explique la loi d'Ohm en une phrase.", backend="ollama", model=test_ollama_model)
        print(f"Réponse Ollama ({test_ollama_model}): \n{response_ollama}")
    except Exception as e:
        print(f"Erreur lors du test Ollama: {e}")
        logger.warning("Le test Ollama a échoué. Vérifiez que Ollama est en cours d'exécution")

    # === Test 2: Gemini simple (texte) ===
    print("\n--- Test 2: Appel Gemini simple (texte) ---")
    try:
        response_gemini_text = call_llm("Quelle est la capitale de l'Australie?", backend="gemini")
        print(f"Réponse Gemini (texte): \n{response_gemini_text}")
    except Exception as e:
        print(f"Erreur lors du test Gemini (texte): {e}")

    # === Test 3: Gemini avec image (API REST forcée si variable définie) ===
    print("\n--- Test 3: Appel Gemini avec image ---")
    # Créer une image simple en base64 pour le test (carré rouge 10x10 px)
    # Vous pouvez remplacer ceci par la lecture d'un vrai fichier image si besoin
    # Note: Une vraie image est préférable pour un test réaliste.
    dummy_image_base64 = "iVBORw0KGgoAAAANSUhEUgAAAAoAAAAKAYAAACNM+s9AAAFU1EQVR42mP8z8AARM="

    # Tentative de lecture d'une image locale si elle existe
    test_image_path = "test_image.png" # Mettez le chemin vers une image PNG de test ici
    try:
        if os.path.exists(test_image_path):
            with open(test_image_path, "rb") as f:
                image_bytes = f.read()
                image_base64_to_use = base64.b64encode(image_bytes).decode('utf-8')
                logger.info(f"Utilisation de l'image de test locale: {test_image_path} ({len(image_base64_to_use)} octets)")
            else:
                logger.warning(f"Image de test locale '{test_image_path}' non trouvée. Utilisation de l'image de test par défaut")
                image_base64_to_use = dummy_image_base64
    except Exception as e:
        logger.error(f"Erreur lors de la lecture de l'image de test locale: {e}. Utilisation de l'image de test par défaut")
        image_base64_to_use = dummy_image_base64

```

```
try:
    if not os.getenv("GEMINI_API_KEY"):
        print("Skipping test Gemini image: Clé API non configurée.")
    else:
        response_gemini_image = call_llm(
            "Décris brièvement cette image.",
            backend="gemini",
            image_base64=image_base64_to_use
        )
        print(f"Réponse Gemini (image):\n{response_gemini_image}")
except Exception as e:
    print(f"Erreur lors du test Gemini (image): {e}")

logger.info("--- Fin des tests llm_utils ---")
```