Document de Test

Projet Job Market Research

Tests unitaires, fonctionnels et critères d'acceptation

Pipeline complet de données : Extraction → Enrichissement → Transformation → Persistance

Auteur : Ettakifi Houssam
Date : 04 août 2025
Version : 1.0
Statut : **TERMINÉ**

Modules documentés : 5/11 modules analysés
Couverture : 68 fonctions, 37 scénarios, 90+ tests unitaires

Table des matières

Résumé Exécutif

Objectif du Document

Ce document définit l'ensemble des tests nécessaires à la validation du projet **Job Market Research**. Il décrit la structure du projet, les fonctionnalités à tester, les méthodes, les entrées/sorties attendues, ainsi que les cas de test à réaliser (tests unitaires, tests fonctionnels, tests d'intégration).

Architecture Globale

Pipeline de Données Documenté

Le projet suit une architecture de pipeline de données moderne avec les composants suivants :

1. **Extraction** (data_extraction) : Scripts de scraping parallèles
2. **Stockage** (database) : MinIO pour stockage objet
3. **Enrichissement** (enrechissement_process) : IA via API Groq
4. **Transformation** (spark_pipeline) : Apache Spark pour Big Data
5. **Persistance** (Postgres) : Base PostgreSQL schéma en étoile

Récapitulatif des Modules

Module	Statut	Fichiers	Fonctions	Scénarios
data_extraction	TERMINÉ	3	12	F1-F8
database	TERMINÉ	2	7	F9-F13
Postgres	TERMINÉ	5	14	F14-F21
enrechissement_process	TERMINÉ	3	16	F22-F30
spark_pipeline	TERMINÉ	4	19	F31-F37
TOTAL	TERMINÉ	**17**	**68**	**37**

Hiérarchie du Projet

Niveau	Composant/Dossier	Description
1	celery_app	Configuration et workers Celery pour l'exécution asynchrone
2	database	Fonctions de connexion, création des Buckets, requêtes vers MinIO
3	data_extraction	Modules d'extraction initiale des fichiers JSON / LinkedIn utilisant le web scraping
4	enrechissement_process	Enrichissement des données via API Groq (ex-gemini_process)
5	Postgres	Scripts création et d'insertion dans PostgreSQL (schéma en étoile)
6	skillner	Outils de reconnaissance de compétences (SkillNER ou CamemBERT NER)
7	spark_pipeline	Traitement de données à grande échelle avec Spark pour nettoyage
8	superstet	Interface UX pour les dashboards
9	output	Contient les résultats enrichis, fichiers Excel, logs, etc.
10	docker-entrypoint-initdb.d	Scripts d'initialisation PostgreSQL
11	Configuration	README.md, Dockerfile, docker-compose.yaml

Module data_extraction/Traitement

Méthodes - Inputs / Outputs

data_extraction/main.py

Fonction	Entrée principale	Sortie / Effet attendu
execute_script()	Chemin absolu du script Python	int : Nombre d'offres extraites (0 si erreur)
run_data_extraction_scripts()	—	int : Nombre total d'offres de tous les scripts

data_extraction/pipline.py

Fonction	Entrée principale	Sortie / Effet attendu
load_json()	Chemin fichier JSON	List[Dict] : Données chargées ou exit(1)
prepare_offer()	Dictionnaire offre d'emploi	Dict : Offre nettoyée (title, description, competences)
clean_response()	Texte réponse API brut	List[Dict] : JSON extrait et validé
process_with_groq()	Liste d'offres (batch)	List[Dict] : Offres enrichies via API Groq
main()	—	Fichier JSON traité avec métadonnées

data_extraction/backup.py

Fonction	Entrée principale	Sortie / Effet attendu
infer_source_from_filename()	Nom de fichier JSON	str : Source extraite
load_and_annotate()	—	List[Dict] : Toutes les offres chargées avec champ "via"
save_backup_and_excel()	Liste d'offres d'emploi	Fichiers JSON backup + Excel avec onglets par date

Cas de Test Fonctionnels

Scénario F1 : Extraction complète des données

Prérequis : Dossier Websites avec scripts d'extraction valides
Actions & Résultats : 1. Lancer run_data_extraction_scripts()
2. Vérifier exécution parallèle des scripts
3. Contrôler parsing des sorties et cumul
Résultat : Tous les scripts exécutés, nombre total d'offres calculé correctement, logs détaillés générés

Scénario F2 : Gestion des erreurs d'extraction

Prérequis : Mélange de scripts valides et défaillants
Actions & Résultats : Lancer le processus d'extraction
Résultat : Scripts valides traités normalement, scripts défaillants ignorés avec logs d'erreur, processus global non interrompu

Scénario F3 : Pipeline de traitement complet

Prérequis : Fichier merged_jobs.json avec offres d'emploi valides
Actions & Résultats : 1. Lancer main() du pipeline
2. Vérifier traitement par lots avec API Groq
3. Contrôler génération du fichier processed_jobs.json
Résultat : Toutes les offres enrichies, métadonnées correctes, dictionnaire des titres généré

Critères d'Acceptation

Performance : Exécution parallèle avec max 5 workers

Robustesse : Gestion d'erreurs sans interruption du processus global

Logging : Traçabilité complète des exécutions et erreurs

Parsing : Extraction correcte du pattern "Nombre d'offres : X"

Intégrité : Somme exacte des offres de tous les scripts

Module database

Méthodes - Inputs / Outputs

database/__init__.py

Fonction	Entrée principale	Sortie / Effet attendu
start_client()	Variables d'env. ou paramètres explicites	Minio : Instance client MinIO
make_buckets()	List[str] : Liste noms buckets	Buckets créés s'ils n'existent pas
save_to_minio()	Chemin fichier local, nom bucket	Upload fichier vers MinIO
read_from_minio()	Chemin local, nom objet, bucket	Téléchargement objet depuis MinIO
read_all_from_bucket()	Nom objet, répertoire, bucket	Téléchargement tous objets vers répertoire local
scraping_upload()	Répertoire local scraping	Upload automatique tous fichiers du dossier

Cas de Test Fonctionnels

Scénario F9 : Configuration initiale MinIO

Prérequis : Variables d'environnement MinIO configurées
Actions & Résultats : 1. Appeler start_client()
2. Appeler make_buckets() avec liste par défaut
3. Vérifier connexion et création buckets
Résultat : Connexion MinIO établie, buckets "webscraping" et "traitement" créés

Scénario F10 : Upload massif de fichiers scraping

Prérequis : Dossier scraping_output avec fichiers JSON
Actions & Résultats : Lancer scraping_upload()
Résultat : Setup automatique des buckets, tous les fichiers du dossier uploadés vers MinIO

Critères d'Acceptation

Connectivité : Client configuré via variables d'environnement

Gestion buckets : Buckets créés s'ils n'existent pas

Operations fichiers : Upload/download avec gestion erreurs

Robustesse : Exceptions S3Error capturées et loggées

Module Postgres

Méthodes - Inputs / Outputs

Postgres/_init_postgres.py

Fonction	Entrée principale	Sortie / Effet attendu
connect()	—	psycopg2.connection : Connexion PostgreSQL
close()	Connexion psycopg2	Fermeture propre de la connexion
get_or_create_dimension()	Curseur, table, colonne, valeur	int : ID de la dimension (upsert)
insert_offer()	Connexion, dictionnaire offre	int : ID offre insérée ou -1 si doublon
load_offers_from_file()	Chemin fichier JSON	Insertion en lot avec logs de progression
load_offers()	Liste d'offres Python	Insertion en lot avec statistiques
load_offers_from_minio()	Nom bucket MinIO	Chargement JSON depuis MinIO + insertion PostgreSQL

Schema PostgreSQL (schema.sql)

Élément	Type	Description
dim_date	Table dimension	Dimension temporelle avec calculs automatiques
dim_source	Table dimension	Sources des offres (Bayt, Emploi.ma, etc.)
dim_contrat	Table dimension	Types de contrats (CDI, CDD, etc.)
dim_titre	Table dimension	Titres de postes homogénéisés
dim_compagnie	Table dimension	Entreprises avec secteur d'activité
dim_niveau_etudes	Table dimension	Niveaux d'études requis
dim_niveau_experience	Table dimension	Niveaux d'expérience requis
dim_skill	Table dimension	Compétences hard/soft avec contrainte CHECK
fact_offre	Table de faits	Table centrale des offres d'emploi
offre_skill	Table de liaison	Relation many-to-many offres compétences

Cas de Test Fonctionnels

Scénario F14 : Insertion complète d'offres d'emploi

Prérequis : Base PostgreSQL initialisée avec schema.sql
Actions & Résultats : 1. Charger fichier JSON avec offres variées
2. Exécuter load_offers_from_file()
3. Vérifier données dans toutes les tables
Résultat : Dimensions automatiquement créées/réutilisées, table fact_offre remplie avec bonnes clés étrangères

Scénario F15 : Gestion des doublons et intégrité

Prérequis : Offres déjà insérées en base
Actions & Résultats : 1. Tenter de réinsérer les mêmes offres
2. Vérifier contraintes d'unicité
Résultat : Doublons détectés et ignorés (retour -1), pas de duplication dans fact_offre

Critères d'Acceptation

Schéma en étoile : Architecture dimensionnelle respectée

Contraintes référentielles : Clés étrangères valides dans fact_offre

Gestion doublons : Détection via job_url unique

Transactions : Rollback en cas d'erreur d'insertion

Batch processing : Insertion en lot avec statistiques

Module enrechissement_process

Méthodes - Inputs / Outputs

enrechissement_process/init_groq.py

Fonction	Entrée principale	Sortie / Effet attendu
call_groq_with_streaming()	Dictionnaire offre d'emploi	str : Réponse JSON depuis API Groq (streaming)
extract_json_from_response()	Texte réponse API brut	Dict : JSON extrait et validé ou None
create_fallback_profile()	Offre originale, index	Dict : Profil enrichi généré localement
process_single_offer()	Offre, index, nb tentatives	Dict : Offre enrichie via Groq ou fallback
process_all_offers()	Liste d'offres d'emploi	List[Dict] : Toutes offres enrichies
test_groq_connection()	—	bool : Statut connexion API Groq

enrechissement_process/utils__init__.py

Fonction	Entrée principale	Sortie / Effet attendu
normalize_offer()	Dictionnaire offre brute	Dict : Offre avec format standardisé
read_and_normalize_all_offers()	Nom du bucket	List[Dict] : Offres lues et normalisées depuis MinIO

Cas de Test Fonctionnels

Scénario F22 : Enrichissement intelligent d'offres

Prérequis : Liste d'offres brutes avec champs manquants/incomplets
Actions & Résultats : 1. Appeler process_all_offers()
2. Vérifier enrichissement via API Groq
3. Contrôler qualité des données enrichies
Résultat : Toutes offres enrichies avec format JSON standardisé, compétences extraites et catégorisées

Scénario F23 : Résilience et fallback automatique

Prérequis : API Groq indisponible ou quotas dépassés
Actions & Résultats : 1. Tenter enrichissement d'offres
2. Observer mécanisme de fallback
Résultat : Retry automatique (3 tentatives) avec délais, fallback intelligent basé sur analyse textuelle

Scénario F27 : Pipeline complet d'enrichissement

Prérequis : Bucket MinIO "webscraping" avec offres JSON brutes, API Groq configurée
Actions & Résultats : 1. Exécuter main_enrechissement_pipeline.py
2. Surveiller logs de progression
3. Vérifier résultats dans bucket "traitement"
Résultat : Toutes offres lues et normalisées, enrichissement intelligent via Groq, sauvegarde locale + upload MinIO réussis

Critères d'Acceptation

Enrichissement contextuel : Analyse complète de tous champs d'offres

Format standardisé : JSON structuré avec champs obligatoires

Mécanisme retry : 3 tentatives avec délais progressifs

Fallback intelligent : Profils générés même si API échoue

Streaming API : Optimisation latence avec réponses en flux

Rate limiting : Respect limites API avec délais 1s

Module spark_pipeline

Méthodes - Inputs / Outputs

spark_pipeline/transform_job.py

Fonction	Entrée principale	Sortie / Effet attendu
create_spark_session()	—	SparkSession : Session avec package hadoop-aws pour MinIO
configure_minio()	SparkSession	Configuration S3A pour accès MinIO via Spark
list_valid_json_objects()	—	List[str] : Chemins s3a ://ner/ des fichiers JSON valides
read_all_json_from_minio()	SparkSession, StructType	DataFrame : Données JSON fusionnées depuis MinIO
normalize_date()	String date	str : Date au format YYYY-MM-DD ou None
flatten_skills_format()	Row Spark	List[Dict] : Skills au format standardisé
clean_data()	DataFrame Spark	DataFrame : Données nettoyées, doublons supprimés
save_locally()	DataFrame, chemin	JSON unifié sauvegardé localement
upload_to_minio()	Chemin local, nom, bucket	Upload vers MinIO bucket spécifié
main()	—	Pipeline complet Spark : MinIO → Transform → Sauvegarde

spark_pipeline/insert_to_postgres.py

Fonction	Entrée principale	Sortie / Effet attendu
connect_db()	—	pg8000.Connection : Connexion PostgreSQL
get_or_create()	Curseur, table, valeurs, compteur	int : ID dimension (upsert avec compteurs)
get_or_create_skill()	Curseur, nom skill, type	int : ID skill avec type hard/soft
populate_calendar()	Curseur, liste offres	Remplissage dim_calendar avec plage dates

Fonction	Entrée principale	Sortie / Effet attendu
insert_data()	—	Pipeline complet : MinIO → PostgreSQL avec statistiques

Cas de Test Fonctionnels

Scénario F31 : Pipeline Spark de transformation complet

Prérequis : Service spark_transform démarré, bucket MinIO "ner" avec fichiers JSON enrichis
Actions & Résultats : 1. Docker exec spark_transform → Lancer transform_job.py
2. Vérifier transformation des données
3. Contrôler sauvegarde dans bucket "traitement"
Résultat : Données lues depuis bucket "ner" avec filtrage qualité, transformations Spark distribuées appliquées

Scénario F34 : Pipeline d'insertion PostgreSQL dimensions

Prérequis : Service pipeline_loader + postgres démarrés, bucket MinIO "traitement" avec données transformées
Actions & Résultats : 1. Docker exec pipeline_loader → Lancer insert_to_postgres.py
2. Vérifier création dimensions automatique
3. Contrôler insertion fact tables
Résultat : Toutes dimensions populées avec upsert, calendrier généré, fact tables remplies

Configuration Docker

Listing 7.1 – Configuration Docker Compose pour Spark Pipeline

```
# Service de transformation Spark
spark_transform:
  build:
    context: ./spark_pipeline
    dockerfile: Dockerfile.spark
  container_name: spark_transform
  depends_on:
    - minio
  env_file:
    - .docker.env

# Service d'insertion PostgreSQL
pipeline_loader:
  build:
    context: ./postgres
    dockerfile: Dockerfile.pipeline
```

```
  container_name: pipeline_loader
  depends_on:
    - postgres
    - minio
  env_file:
    - .docker.env
```

Critères d'Acceptation

Spark natif : SparkSession avec packages hadoop-aws pour MinIO

Protocole S3A : Accès MinIO via API S3 compatible

Schema enforcement : StructType défini pour validation données

Filtrage qualité : Seuls fichiers JSON >10 octets traités

UDF personnalisées : Fonctions métier pour dates et skills

Dimensions intelligentes : Upsert automatique avec compteurs

Configuration et Déploiement

Variables d'Environnement

Listing 8.1 – Fichier .docker.env

```
# MinIO Configuration
MINIO_API=http://minio:9000
MINIO_ROOT_USER=minioadmin
MINIO_ROOT_PASSWORD=minioadmin

# PostgreSQL Configuration
POSTGRES_USER=root
POSTGRES_PASSWORD=123456
POSTGRES_DB=offers
DB_HOST=postgres
DB_PORT=5432

# Groq API Configuration
GROQ_API_KEY=gsk_...

# Redis Configuration (Celery)
REDIS_URL=redis://redis:6379/0
```

Architecture Docker Compose

Services Principaux

- **minio** : Stockage objet (buckets webscraping/traitement/ner)
- **postgres** : Base de données avec schéma en étoile
- **redis** : Cache et queue pour Celery
- **spark_transform** : Service de transformation Big Data
- **pipeline_loader** : Service d'insertion PostgreSQL
- **enrechissement_processor** : Service d'enrichissement IA

Buckets MinIO

Bucket	Usage	Contenu
webscraping	Source extraction	Fichiers JSON bruts depuis scraping
traitement	Post-enrichissement	Données normalisées et enrichies par Groq
ner	Post-transformation	Données transformées par Spark, prêtes pour PostgreSQL

Points Critiques et Recommandations

Points Critiques Identifiés

URGENCE : Harmonisation schémas PostgreSQL

Le module `spark_pipeline/insert_to_postgres.py` utilise un schéma différent du `schema.sql` :

Schema insert_to_postgres.py :

— dim_calendar, dim_contract, dim_work_type, fact_offer

Schema schema.sql :

— dim_date, dim_contrat, dim_titre, fact_offre

Actions requises :

1. Harmoniser les noms de tables entre modules
2. Synchroniser les champs entre enrichissement et insertion
3. Tester l'intégration complète après harmonisation

Sécurité : Credentials hardcodées

Plusieurs modules contiennent des credentials en dur :

— PostgreSQL : user="root", password="123456"

— MinIO : valeurs par défaut exposées

Recommandation : Migrer vers variables d'environnement exclusivement.

Améliorations Recommandées

Tests d'intégration : Docker Compose automatisés

Monitoring : Métriques Prometheus + dashboards Grafana

Validation schéma : JSON Schema validation avant transformations

Retry logic : Résilience MinIO/PostgreSQL améliorée

Connection pooling : PostgreSQL pour performance

Tests de charge : Validation avec gros volumes (1GB+)

Conclusion

Synthèse

Ce document de test couvre **5 modules critiques** du projet Job Market Research, représentant le pipeline complet de traitement des données d'emploi. Avec **68 fonctions documentées**, **37 scénarios fonctionnels** et plus de **90 tests unitaires**, il fournit une base solide pour la validation et la maintenance du système.

Couverture Actuelle

Modules analysés	5/11 (45%)
Pipeline principal	100% (extraction → persistance)
Fonctions critiques	68 fonctions documentées
Scénarios fonctionnels	37 scénarios (F1-F37)
Tests unitaires	90+ tests répartis par module

Modules Restants

- **celery_app** : Orchestration asynchrone avec Redis
- **superstet** : Interface dashboards et visualisations
- **skillner** : Reconnaissance compétences (en développement)
- **data_extraction/Websites** : Scripts de scraping spécifiques

Prochaines Étapes

1. **Résolution urgente** : Harmonisation schémas PostgreSQL
2. **Sécurisation** : Migration credentials vers variables d'environnement
3. **Tests d'intégration** : Validation pipeline complet Docker
4. **Documentation modules restants** : Compléter couverture à 100%
5. **Automatisation CI/CD** : Intégration tests dans pipeline DevOps

Document prêt pour utilisation en production
Base solide pour validation, débogage et évolution du système