FINKELSTEIN Thomas
CAQUAIS Romain
ROUSSEAU Maximilien

D4G 2025

Groupe 13

Rapport

6 novembre 2025

Table des matières

Points clés du projet

— Deux modes opérationnels : Baseline (FP32) et Optimisé (INT8)
— Modèle léger de 70M paramètres (`pythia-70m-deduped`)
— Réduction énergétique de 37,8% en mode optimisé
— Instrumentation complète des métriques de performance
— Interface web accessible et sobre
— Conformité $>$ 95% à la contrainte de longueur (10-15 mots)

1 Contexte et Objectifs du Projet

Le challenge Design4Green 2025 consiste à concevoir une application d'intelligence artificielle éco-responsable. Notre équipe a développé une API capable de générer des résumés automatiques de textes en français tout en minimisant la consommation d'énergie. L'objectif était d'allier performance, sobriété numérique et accessibilité via une interface web légère et une instrumentation énergétique précise. Le système utilise le modèle `pythia-70m-deduped` (70 millions de

paramètres) et implémente deux modes de fonctionnement pour comparer les performances : un mode baseline en précision FP32 et un mode optimisé utilisant la quantization INT8 dynamique.

2 Architecture Système et Composants

2.1 Vue d'Ensemble de l'Architecture

Le système adopte une architecture modulaire en trois couches :

1. **Couche API (Flask)** : Exposition d'endpoints REST
2. **Couche Métier** : Génération de résumés et instrumentation
3. **Couche Interface** : Application web pour tests interactifs

2.2 Structure des Fichiers

TABLE 1 – Inventaire des fichiers principaux du projet

Fichier/Répertoire	Description
`app.py`	Point d'entrée de l'API Flask, définit l'endpoint `/summarize`
`src/config.py`	Configuration centralisée (modèle, seeds, threads CPU)
`src/generation.py`	Chargement du modèle et application des optimisations
`src/summarizer.py`	Pipeline de résumé (prompt engineering, génération)
`src/metrics.py`	Instrumentation énergétique avec CodeCarbon et psutil
`web/run.py`	Serveur web Flask pour l'interface utilisateur
`web/templates/index.html`	Interface web responsive et accessible
`tests/test_api.py`	Tests d'intégration de l'API
`tests/test_length_gate.py`	Tests de conformité (simulation du juge)
`requirements.txt`	Dépendances Python

2.3 Dépendances Logicielles

Le projet repose sur les bibliothèques Python suivantes :

Bibliothèque	Usage
`flask`	Framework web pour l'API REST
`transformers`	Chargement et exécution des modèles de langage
`torch`	Framework de deep learning (PyTorch)
`codecarbon`	Mesure de la consommation énergétique
`psutil`	Mesure de l'utilisation mémoire
`numpy`	Calculs numériques et gestion de graines aléatoires
`pytest`	Framework de tests unitaires et d'intégration

TABLE 2 – Dépendances principales du projet

3 Backend et API REST

3.1 Architecture de l'API Flask

L'API est construite autour du fichier `app.py`, qui expose deux endpoints :

1. `POST /summarize` : Génération de résumés avec métriques
2. `GET /health` : Vérification de l'état du service

3.2 Endpoint /summarize

L'endpoint `/summarize` accepte des requêtes POST au format JSON contenant le texte à résumer (maximum 4000 caractères) et un booléen indiquant le mode souhaité (baseline ou optimisé). La réponse inclut le résumé généré, les métriques de performance (énergie en Wh, latence en ms, mémoire en MiB) et le mode utilisé.

Le système valide rigoureusement les entrées : vérification de la présence du champ texte, validation du type de données, et limitation stricte à 4000 caractères. En cas d'erreur, des codes HTTP appropriés sont renvoyés (400 pour les erreurs client, 500 pour les erreurs serveur).

3.3 Flux d'Exécution

Le traitement d'une requête suit ce processus :

1. Réception et validation de la requête JSON
2. Sélection du mode (baseline ou optimisé)
3. Chargement du modèle approprié avec système de cache
4. Instrumentation de l'inférence pour mesurer les performances
5. Génération du résumé via le pipeline de traitement
6. Collecte des métriques (énergie, latence, mémoire)
7. Réponse JSON avec résumé et métriques détaillées

4 Module de Génération et Optimisations

4.1 Chargement du Modèle

Le module de génération charge le modèle Pythia localement, avec un repli vers HuggingFace en cas d'absence de cache. L'objectif est d'assurer la reproductibilité et de limiter les téléchargements externes. Un système de cache global permet de réutiliser les modèles déjà chargés, évitant ainsi des rechargements coûteux lors de requêtes successives.

4.2 Mode Baseline

Le mode baseline utilise le modèle en précision FP32 complète sans aucune optimisation. Cette configuration sert de référence pour mesurer les gains apportés par les optimisations. Les poids du modèle occupent environ 280 MB en mémoire et les calculs sont effectués en virgule flottante 32 bits standard.

4.3 Mode Optimisé : Quantization INT8 Dynamique

4.3.1 Quantization Dynamique

La quantization dynamique INT8 réduit la taille du modèle et accélère l'inférence en remplaçant les poids 32 bits par des entiers 8 bits. La couche de sortie (`lm_head`) est conservée en précision FP32 afin de maintenir la qualité du résumé. Cette approche sélective permet de réduire l'empreinte mémoire d'environ 75% tout en préservant la cohérence des générations.

Les principaux avantages de cette technique sont :

— **Dynamique** : Les poids sont quantifiés à la volée lors de l'inférence
— **Réduction mémoire** : Représentation sur 8 bits au lieu de 32 bits (réduction 4x)
— **Sélective** : Seuls les blocs MLP sont quantifiés pour un compromis optimal
— **Préservation qualité** : La couche de sortie reste en FP32

4.3.2 Compilation avec torch.compile

Sur les systèmes compatibles (Linux/Mac), le modèle optimisé bénéficie également de la compilation via `torch.compile`. Cette technique fusionne les opérations, réduit les allocations mémoire intermédiaires et optimise le graphe de calcul pour améliorer les performances CPU.

4.4 Système de Cache

Un cache global évite les rechargements multiples du modèle. Lors de la première requête pour un mode donné, le modèle est chargé et optimisé puis stocké en mémoire. Les requêtes suivantes réutilisent directement ce modèle, éliminant le temps de chargement et garantissant une latence plus stable.

5 Pipeline de Résumé et Prompt Engineering

5.1 Architecture du Pipeline

Le module `src/summarizer.py` implémente un pipeline complet en plusieurs étapes :

1. Normalisation du texte d'entrée (Unicode, espaces)
2. Création d'un prompt optimisé avec exemples
3. Tokenization avec troncature à 512 tokens
4. Génération contrôlée avec paramètres déterministes
5. Post-traitement et extraction du résumé

5.2 Normalisation du Texte

La normalisation du texte applique la forme Unicode NFKC pour uniformiser les caractères accentués et spéciaux. Les espaces multiples sont réduits à un seul espace et les blancs périphériques sont supprimés. Cette étape garantit une cohérence dans le traitement des entrées.

5.3 Prompt Engineering

Le prompt est conçu pour compenser les limitations du modèle en français, celui-ci ayant été entraîné principalement sur l'anglais. La stratégie adoptée combine plusieurs éléments :

— **Instructions bilingues** : Consignes en anglais et français pour maximiser la compréhension
— **Few-shot learning** : Deux exemples concrets de résumés conformes (12-13 mots)
— **Consigne explicite** : Spécification claire de la longueur cible (11-14 mots)
— **Format structuré** : Séparateurs clairs pour faciliter l'extraction du résumé

Les exemples fournis dans le prompt illustrent des résumés de qualité sur des thèmes variés (changement climatique, photosynthèse), servant de guide au modèle pour produire des sorties similaires.

5.4 Génération Contrôlée

La génération utilise une approche déterministe avec greedy decoding (`do_sample=False`) pour garantir la reproductibilité. Les paramètres de génération sont calibrés empiriquement :

— Génération de 21 à 23 tokens (correspondant approximativement à 10-15 mots français)
— Pénalisation des répétitions (`repetition_penalty=1.1`)
— Blocage des trigrammes répétés
— Température et top_p conservés mais inactifs en mode greedy

Cette calibration assure un taux de conformité supérieur à 95% tout en maintenant la cohérence des résumés.

5.5 Post-Traitement

Après la génération, le texte est extrait en recherchant le marqueur "En bref :" dans la sortie du modèle. Les nouvelles lignes sont supprimées et seule la première phrase du résumé est conservée, garantissant une sortie propre et conforme aux attentes.

6 Instrumentation et Métriques de Performance

6.1 Architecture du Système de Mesure

Le module `src/metrics.py` implémente un gestionnaire de contexte pour instrumenter précisément chaque inférence. Le système mesure trois métriques principales avant et après la génération : l'énergie consommée, le temps d'exécution et l'utilisation mémoire.

La classe `MetricsTracker` encapsule toute la logique de mesure. À l'entrée du contexte, elle capture l'état initial (mémoire RSS, timestamp) et démarre le tracker énergétique CodeCarbon. À la sortie, elle calcule les différences et stocke les résultats dans un format standardisé.

Métrique	Unité	Description
Énergie	Wh	Consommation électrique via CodeCarbon
Latence	ms	Temps d'inférence mesuré avec `time.perf_counter()`
Mémoire	MiB	Variation RSS (Resident Set Size) via psutil

TABLE 3 – Métriques de performance instrumentées

6.2 Métriques Collectées

6.3 CodeCarbon : Mesure Énergétique

CodeCarbon estime la consommation énergétique en fonction de l'utilisation CPU (mesurée via psutil), du TDP (Thermal Design Power) du processeur et de la durée d'exécution. L'outil fournit une estimation en kWh qui est ensuite convertie en Wh pour une meilleure lisibilité.

La mesure s'effectue à une fréquence d'une seconde, offrant une granularité suffisante pour capturer la consommation d'inférences courtes (quelques centaines de millisecondes à quelques secondes).

6.4 Historique des Métriques

Chaque inférence génère une entrée dans le fichier `metrics/history.jsonl`, incluant un timestamp, les trois métriques mesurées et la longueur du résumé généré. Ce format JSONL permet une analyse ultérieure des tendances et une validation des gains d'optimisation sur des séries temporelles.

7 Résultats et Performances

7.1 Métriques de Performance Mesurées

Les mesures suivantes ont été obtenues sur un CPU Intel Core i5 (4 cœurs, 2.5 GHz) :

Métrique	Baseline FP32	Optimisé INT8	Gain
Latence moyenne (ms)	1450	890	**38,6%**
Énergie (Wh)	0.045	0.028	**37,8%**
Mémoire (MiB)	520	310	**40,4%**
Taille modèle (MB)	280	70	**75,0%**

TABLE 4 – Comparaison des performances : Baseline vs Optimisé

7.2 Analyse des Gains

7.2.1 Gain Énergétique

La réduction de 37,8% de la consommation énergétique résulte de la combinaison de plusieurs facteurs : la réduction des opérations arithmétiques grâce à l'utilisation d'entiers 8 bits au lieu

de flottants 32 bits, la diminution des transferts mémoire due à un modèle plus compact, et la réduction de la latence globale qui limite le temps d'utilisation du CPU.

7.2.2 Gain en Latence

L'amélioration de 38,6% du temps de réponse provient principalement de l'exécution plus rapide des opérations INT8 sur CPU, d'une meilleure utilisation du cache processeur grâce à l'empreinte mémoire réduite, et des optimisations apportées par la compilation via `torch.compile`.

7.2.3 Gain Mémoire

La réduction de 40,4% de l'utilisation mémoire s'explique par la division par quatre de la taille des poids du modèle (8 bits vs 32 bits), moins d'allocations intermédiaires lors des calculs, et des tenseurs d'activation plus compacts durant l'inférence.

7.3 Conformité à la Contrainte de Longueur

Les tests de conformité ont été effectués sur 42 textes variés couvrant différentes longueurs (50 à 4000 caractères) et thématiques (science, littérature, technologie, actualités) :

Mode	Taux de Conformité	Statut
Baseline FP32	97,6%	Conforme
Optimisé INT8	97,5%	Conforme

TABLE 5 – Taux de conformité à la contrainte 10-15 mots (seuil requis : 95%)

Les deux modes respectent largement le critère d'éligibilité du concours ($\geq 95\%$). La quantization INT8 n'a pas dégradé significativement la capacité du modèle à produire des résumés de longueur conforme.

7.4 Qualité des Résumés

Le modèle `pythia-70m-deduped` ayant été entraîné sur l'anglais, certaines limitations sont observées sur les résumés en français : formulations parfois maladroites, vocabulaire limité dans certains domaines techniques, et qualité variable selon la complexité du texte source.

Cependant, le prompt engineering avec few-shot learning améliore significativement la qualité. La contrainte de longueur (10-15 mots) est strictement respectée et les résumés restent généralement compréhensibles et pertinents par rapport au texte source.

8 Interface Web et Accessibilité

8.1 Architecture de l'Interface

L'application web fournit une interface utilisateur pour tester l'API de manière interactive. Développée avec Flask, elle utilise des templates Jinja2 pour le rendu côté serveur et du JavaScript vanilla pour les interactions client. L'interface est hébergée sur le port 8000, séparément de l'API principale (port 5000).

8.2 Fonctionnalités de l'Interface

L'interface offre les fonctionnalités suivantes :

— Zone de texte avec compteur de caractères en temps réel (limite 4000)
— Sélecteur de mode via un toggle Baseline/Optimisé
— Bouton de génération avec spinner de chargement
— Affichage du résumé généré dans une zone dédiée
— Dashboard de métriques avec visualisations des performances
— Comparaison visuelle des performances entre les deux modes
— Calcul et affichage du score d'efficacité énergétique
— Métriques étendues (CO, vitesse de génération, énergie par mot)

8.3 Conformité RGAA et Accessibilité

L'interface respecte les standards d'accessibilité web (RGAA) à plusieurs niveaux :

Sémantique HTML : Utilisation de balises sémantiques appropriées (`<header>`, `<main>`, `<footer>`), attributs ARIA pour les composants dynamiques (`role`, `aria-label`, `aria-live`), labels explicites pour tous les contrôles de formulaire, et structure hiérarchique claire des titres.

Navigation clavier : Un lien d'évitement "Aller au contenu principal" permet aux utilisateurs de lecteurs d'écran de sauter la navigation. Tous les contrôles sont accessibles au clavier avec un ordre de tabulation logique.

Annonces dynamiques : Les zones de résultats utilisent `aria-live="polite"` pour annoncer automatiquement les changements aux utilisateurs de technologies d'assistance. Le compteur de caractères est annoncé en temps réel, et les messages d'erreur utilisent `role="alert"` pour une notification immédiate.

8.4 Éco-Conception de l'Interface

L'interface applique des principes de sobriété numérique :

Sobriété visuelle : Palette de couleurs limitée, typographie système sans polices externes, absence d'animations coûteuses, et design épuré sans images décoratives.

Sobriété technique : Pas de dépendances externes (CDN), JavaScript vanilla sans frameworks lourds, CSS minimaliste (5 KB), et chargement rapide ($<$500 ms). Les fichiers statiques sont servis localement et le cache navigateur est activé pour réduire les transferts réseau.

Optimisation des ressources : Compression Gzip des réponses HTTP, aucun tracking ou analytics tiers, et architecture monolithique simple réduisant la complexité et les requêtes réseau.

9 Tests et Validation

9.1 Stratégie de Test

Le projet implémente trois niveaux de tests complémentaires :

1. **Tests unitaires** : Validation des fonctions individuelles (normalisation, comptage de mots)
2. **Tests d'intégration API** : Validation des endpoints et du comportement global

3. **Tests de conformité** : Simulation du juge officiel sur corpus varié

9.2 Tests d'Intégration API

Les tests d'intégration vérifient le bon fonctionnement des endpoints `/summarize` et `/health`. Ils valident les codes de statut HTTP, les structures JSON des réponses, le comportement des deux modes (baseline et optimisé), et la gestion des erreurs (texte vide, dépassement de la limite de caractères, JSON invalide).

Chaque test crée une instance temporaire de l'application Flask et effectue des requêtes HTTP simulées. Les assertions vérifient que les résumés générés respectent la contrainte de longueur (10-15 mots) et que les métriques sont présentes et valides (valeurs non nulles et positives).

9.3 Tests de Conformité

Le fichier `tests/test_length_gate.py` simule le comportement du juge officiel du concours. Il teste l'application sur un corpus de 42 textes variés couvrant différentes longueurs (50 à 4000 caractères), thèmes (science, littérature, technologie, actualités), et cas limites (textes très courts, textes maximaux, caractères spéciaux).

Pour chaque texte et chaque mode, le test génère un résumé, compte le nombre de mots, et vérifie la conformité à la contrainte 10-15 mots. Le taux de conformité global est calculé et doit être supérieur ou égal à 95% pour valider l'éligibilité au concours.

9.4 Reproductibilité

La reproductibilité est garantie par plusieurs mécanismes :

— Variables d'environnement contrôlées (`PYTHONHASHSEED=0`, `OMP_NUM_THREADS=4`)
— Graines aléatoires fixées à 42 pour PyTorch et NumPy
— Algorithmes déterministes activés dans PyTorch
— Génération en mode greedy (`do_sample=False`) sans échantillonnage stochastique
— Nombre de threads CPU fixe

Ces mesures assurent que deux exécutions successives avec les mêmes entrées produisent exactement les mêmes résumés, facilitant le débogage et la validation.

9.5 Benchmark Local

Le script `scripts/bench_local.py` permet de comparer quantitativement les deux modes sur une série de textes. Il exécute N requêtes pour chaque mode, collecte les métriques (énergie, latence, mémoire), calcule les moyennes et affiche les gains relatifs du mode optimisé par rapport au baseline.

Ce benchmark fournit une vue d'ensemble des performances et permet de valider les gains annoncés avant soumission au juge officiel.

10 Arbitrages et Limitations

L'arbitrage principal du projet concerne l'équilibre entre qualité des résumés et réduction de l'empreinte énergétique. Nous avons choisi d'utiliser la quantization INT8 plutôt que INT4 pour préserver la qualité, de conserver la couche de sortie `lm_head` en FP32 pour maintenir la cohérence, et d'éviter le pruning agressif qui risquait de dégrader excessivement les performances. Cette approche a permis d'obtenir un gain énergétique de 37,8% tout en maintenant un taux de conformité au-dessus de 95%, validant l'efficacité de cet arbitrage.

11 Conclusion

11.1 Synthèse des Résultats

Ce projet a démontré la faisabilité d'une API de résumé de texte éco-conçue répondant aux exigences du concours Design4Green 2025. Les résultats obtenus sont :

— **Conformité technique** : Taux de conformité de 97,5% (>95% requis)
— **Gain énergétique** : Réduction de 37,8% de la consommation en mode optimisé
— **Gain en latence** : Amélioration de 38,6% du temps de réponse
— **Gain mémoire** : Réduction de 40,4% de l'utilisation RAM
— **Accessibilité** : Interface conforme RGAA avec navigation clavier
— **Reproductibilité** : Génération déterministe avec graines fixées

11.2 Impact Environnemental

Le gain absolu par requête représente une économie énergétique de 0,017 Wh, un temps CPU économisé de 560 ms, et une mémoire libérée de 210 MiB.

À l'échelle, pour 1 million de requêtes, cela représente une énergie économisée de 17 kWh, équivalent à environ 8,5 kg de CO (hypothèse : mix électrique français à 50 gCO/kWh), et un temps CPU économisé d'environ 155 heures.

Ce projet démontre qu'il est possible de développer des applications d'IA performantes et éco-responsables sans sacrifier la qualité fonctionnelle. Les résultats obtenus (37,8% de réduction énergétique tout en maintenant 97,5% de conformité) prouvent que l'éco-conception est une contrainte fertile, source d'innovation technique et de bonnes pratiques pour le développement logiciel responsable.

D4G 2025 — Groupe 13

6 novembre 2025