GreenFaaS : La motivation du projet

L'idée du projet GreenFaaS, est de proposer **une nouvelle conception des plateformes de services FaaS (Function as a Service) permettant de réduire la consommation énergétique lors de l'exécution des fonctions tout en conservant autant que possible les performances (temps d'exécution ou qualité du résultat)**. La première étape du projet est de montrer que pour une tâche (une fonction), il existe des implémentations alternatives dont l'exécution peut produire des résultats différents mais acceptables, avec des consommations énergétiques et consommations de ressources différentes. Pour cela, nous avons implémenté des benchmarks (*step functions* et *single function*) de l'état de l'art, en implémentant et en comparant différentes alternatives.

Les expérimentations ont été réalisées sur la plate-forme **OpenWhisk**, en utilisant **l'instance r320 du cluster APT de la plate-forme cloudlab**. Comme service de stockage distant, nous avons openstack swift. Les benchmark ayant été implémenté sont les suivants :

A. thumbnail

Le benchmark télécharge une image depuis le stockage cloud, la redimensionne à la taille d'une vignette, upload ensuite la nouvelle version plus petite de l'image. Pour les expérimentations, nous avons sélectionné des images de tailles différentes provenant du jeu de données image-net utilisé dans le papier EcooFaaS . Il contient au total plus 1 millions d'images, et grâce au cdf à figure 1, on peut voir que les images dominante dans le jeu de données sont celles dont la taille est inférieure à 0.256 Mb soit 256 Kb. Nous avons donc sélectionné 05 images respectivement de taille 15 Mb (la plus grande), 1 Mb, 256 Kb, 100 Kb (la taille moyenne), et 500 b (la plus petite).

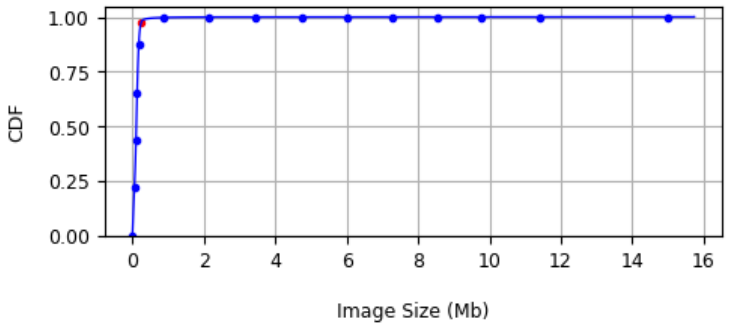

fig. 1: CDF

Nous avons comparé 04 implémentations alternatives pour ce benchmark. Chaque alternative correspond à l'utilisation différente d'une bibliothèque python (opencv, pilow, wand, pygame) pour redimensionner l'image. Pour chaque bibliothèque avec chacune des images nous avons effectué **100 exécutions** successives et les métriques collectés ont ensuite été normalisées de tel sorte que les valeurs les plus petites soient plus proches de 0 et les plus grandes plus

proches de 1. Les observations et interprétations faites après les expérimentations sont les suivantes :

Observation 1: **<u>La bibliothèque pillow a des temps de traitement les plus bas</u>**
L'histogramme à la figure 2 présente les coûts moyens en temps de l'opération de redimensionnement avec les différentes bibliothèques utilisées pour chaque taille image.

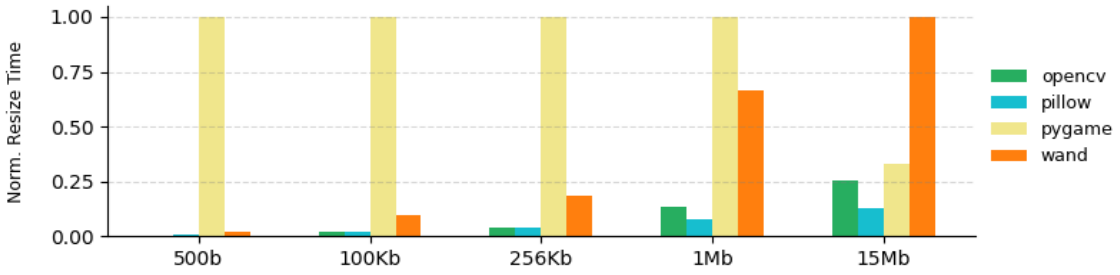

fig. 2: Temps moyen de redimensionnement

On observe que, pour des images de taille comprise entre 500b et 1 Mb, le temps de traitement avec la bibliothèque Pygame est largement supérieur par rapport aux autres bibliothèques. Toutefois, l'écart entre Pygame et Wand se réduit à mesure que la taille de l'image augmente, au point où Wand est la bibliothèque la plus lente pour l'image de 15 Mb. On conserve, en revanche, des temps de traitement les plus bas avec pillow.

Observation 2: <u>La bibliothèque pillow conservent des coûts en temps total légèrement inférieurs aux autres bibliothèques</u>
Nous avons ensuite comparé grâce à l'histogramme de la figure 3 le coût moyen total en temps nécessaire pour l'exécution du benchmark, incluant le temps de redimensionnement, de téléchargement, et retour de l'image.

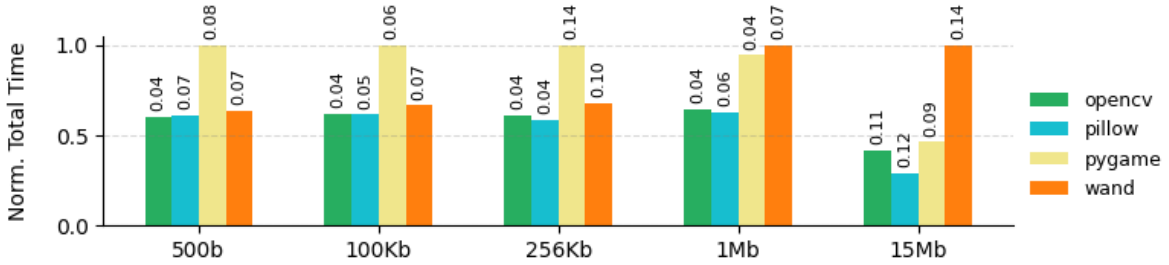

fig. 3: Temps moyen total

Au sommet de chaque barre de l'histogramme est marqué l'écart type de l'ensemble des exécutions pour avoir une indication sur la marge d'erreur lors des expérimentations. En observant l'histogramme, nous constatons que Pillow conserve néanmoins des coûts totaux en temps inférieurs par rapport aux autres.

Observation 3: <u>Pillow et OpenCV ont des consommations énergétiques plus basses que les autres bibliothèques.</u>

Nous nous sommes ensuite intéressés à l'énergie consommée lors de l'exécution du benchmark. L'histogramme à la figure 4 permet de comparer l'énergie moyenne consommée entre les différentes bibliothèques. On relève principalement que, les bibliothèques pillow et OpenCV ont une consommations énergétiques plus basses que les autres.

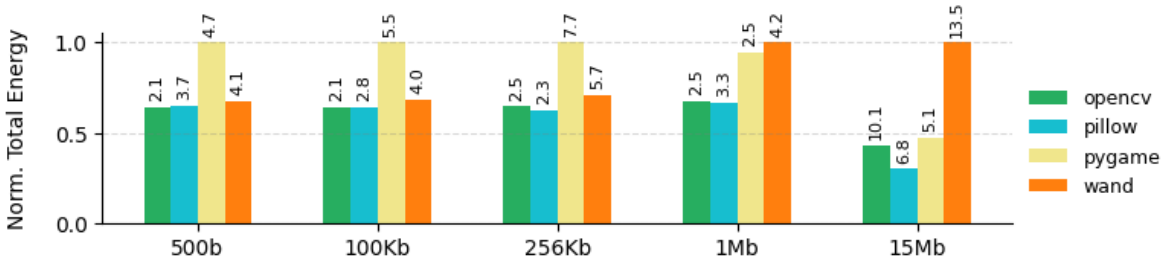

fig. 4: Énergie moyenne consommée

Observation 4 : **<u>OpenCV et Wand produisent des images finales de meilleure qualité</u>**.
Pour finir nous avons comparé la qualité de l'image produite par chaque bibliothèque en se basant sur la taille du fichier final. L'histogramme à la figure 5 ci-dessous montre que, Pour des images les plus représentatives, c'est-à-dire celles dont la taille est inférieur à 256 Kb, la bibliothèque OpenCV produit une image de meilleure qualité. Pour des images de taille plus grande, Wand produit un meilleur résultat.

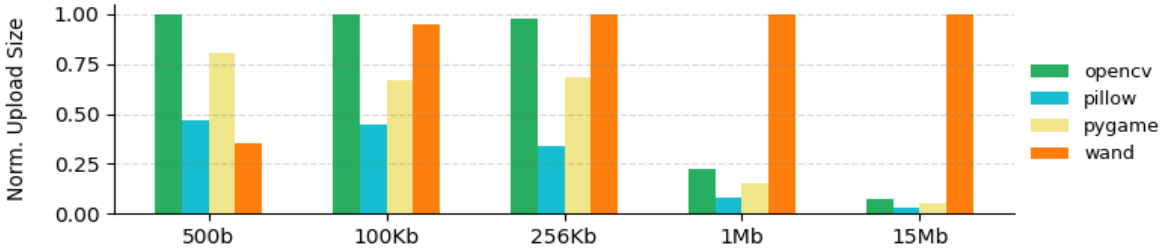

fig. 5: Taille de l'image en sortie

Interprétation côté utilisateur: En tenant compte des analyses effectuées pour ce benchmark, du point de vue de l'utilisateur on peut dire que le choix d'une alternative dépendra de la taille de l'image d'origine. Par exemple, pour des images de tailles inférieures à 256 Kb, la bibliothèque opencv se présente comme le meilleur choix car elle produit une image finale de meilleure qualité, avec un temps d'exécution total légèrement supérieur à pillow et plus bas que les autres. Pour des images de taille plus grande, la bibliothèque wand est le meilleur choix car elle produit une image finale considérablement de meilleure qualité avec un temps d'exécution total légèrement supérieur aux autres bibliothèques.

Interprétation côté fournisseur: Du côté fournisseurs, la meilleure alternative serait d'utiliser la bibliothèque Pillow, car elle permet d'avoir pour toutes tailles d'image, le coût total en temps d'exécution le plus bas et des consommations énergétiques les plus basses bien

que très variables.

Concernant la motivation, est-ce intéressant ? En ce qui concerne les objectifs du projet GreenFaaS, pour ce premier benchmark, on constate qu'il existe effectivement des implémentations alternatives avec des coûts en temps différents, des consommations énergétiques différentes, et des résultats différents mais acceptables.

B. video-processing

Ce benchmark télécharge une vidéo depuis le stockage cloud, la transforme en image au format GIF, et retourne ensuite le résultat. Pour les expérimentations, nous avons sélectionné des vidéos provenant du jeu de données UCF 101 videos également utilisé dans le papier EcoFaaS. Le jeu de données contient 13320 vidéos et à partir du CDF à la figure 6 on constate que les vidéos dominante sont celles dont la taille est inférieure à 1.5 Mo. Nous avons alors sélectionné 04 vidéos respectivement de tailles 6 Mo, 1 Mo, 540 Kb, et 36 Kb.

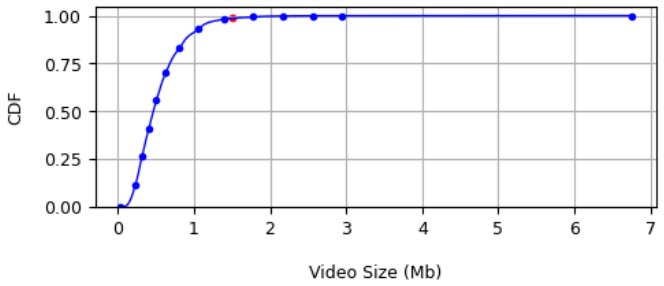

fig. 6: CDF

Nous avons par la suite implémenté et comparé 04 alternatives pour ce benchmark, chacune avec 04 bibliothèques différentes (ffmpeg, imageio, moviepy, opencv) pour transformer une vidéo en GIF. Pour chaque bibliothèque comme précédemment, avec chaque vidéo, nous avons effectué 100 exécutions successives. Les observations et interprétations faites après les expérimentations sont les suivantes :

Observation 1 : ffmpeg a le temps de traitement le plus bas.
Le diagramme à la figure 7 présente les temps moyens nécessaires pour transformer une vidéo en GIF avec chaque bibliothèque. On note principalement que, ffmpeg a le temps de traitement le plus bas peu importe la taille de la vidéo en entrée. L'écart entre les autres bibliothèques n'est pas très grand, mais OpenCV est tout de même assez plus rapide que imageio et moviepy.

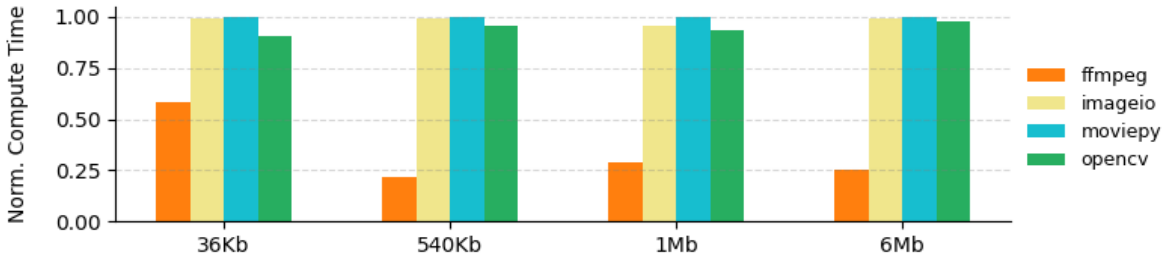

fig. 7: Temps moyen de transformation en GIF

Observation 2: <u>ffmpeg conserve le coût en temps total le plus bas</u>.

L 'histogramme à la figure 8 présente les coûts moyens total en temps pour l'exécution du benchmark, incluant le temps de transformation de la vidéo en GIF, de téléchargement, et retour du résultat. On constate principalement que ffmpeg conserve un coût total en temps plus bas peu importe la vidéo en entrée.

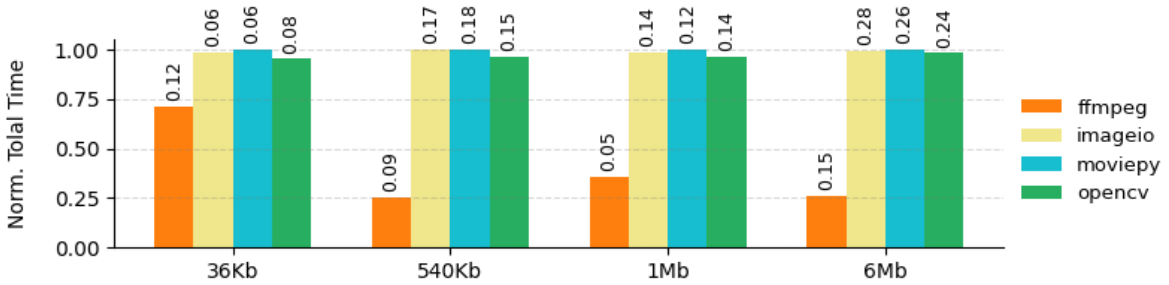

fig. 8: Temps moyen total

Observation 3: <u>ffmpeg a des consommations énergétiques les plus basses.</u>

L'histogramme à la figure 9 présente l'énergie moyenne consommée par chaque bibliothèque avec des vidéos de différentes tailles en entrée. On note principalement que ffmpeg a la consommation énergétique la plus basse quelque soit la vidéo en entrée.

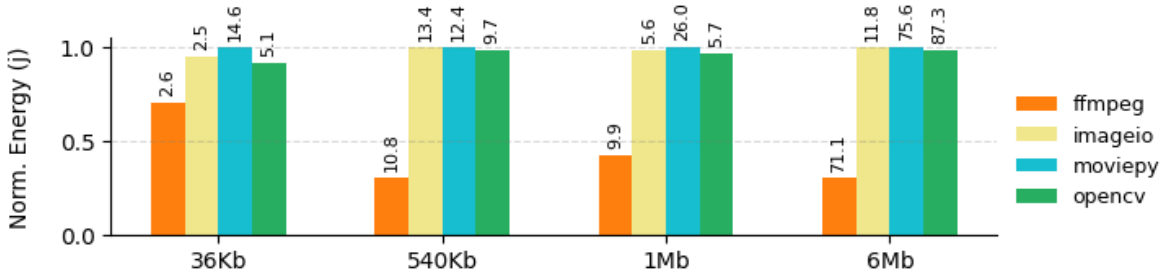

fig. 9: Énergie moyenne consommée

Observation 4: <u>ffmpeg produit une image GIF de moins bonne qualité</u>.

L 'histogramme à la figure 10 permet de comparer la qualité de l'image GIF finale produite par chaque bibliothèque en se basant sur la taille du fichier final. On constate que ffmpeg produit une image GIF de moins bonne qualité que les autres bibliothèques. Excepté ffmpeg les autres bibliothèques produisent en sortie une image GIF de même tailles et donc de qualité

égale.

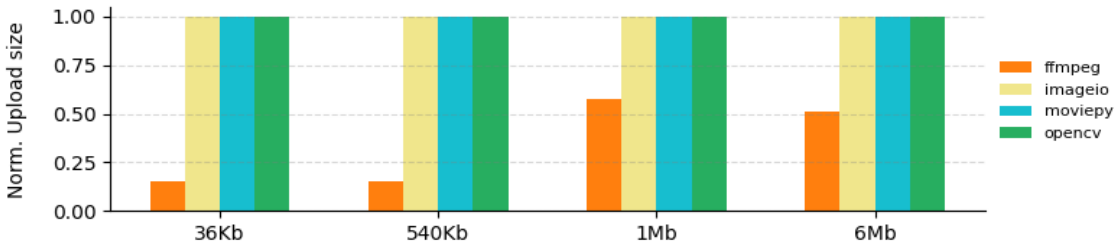

fig. 10: Taille de l'image GIF en sortie

Interprétation côté utilisateur : Sous la base des observations précédentes, du point de vue utilisateur, la meilleure alternative pour ce benchmark serait d'utiliser OpenCV car il produit un résultat de meilleure qualité, avec des coûts en temps globalement plus proches de ffmpeg.

Interprétation côté fournisseur : Pour le fournisseur, la meilleure alternative serait d'utiliser ffmpeg car il a des coûts en temps les plus bas, ainsi que des consommations énergétiques les plus basses.

Concernant la motivation, est-ce intéressant ? En ce qui concerne les objectifs du projet GreenFaaS, on constate là encore qu'il existe pour une tâche précise des implémentations alternatives avec des coûts en temps différents, des consommations énergétiques différentes, et des résultats différents mais acceptables.

C. image-recognition

Le benchmark exécute une tâche de reconnaissance d'image. Il commence par télécharger une image depuis le stockage cloud, puis la soumet en entrée à un modèle de deep learning (ResNet) conçu pour la reconnaissance d'images. Pour ce benchmark, comme précédemment, nous avons sélectionné des images provenant du jeu de données ImageNet. Comme implémentation alternative, nous avons considéré différentes versions du modèle ResNet provenant de la bibliothèque pytorch, à savoir ResNet18, ResNet34, ResNet50, ResNet152 désignant respectivement des versions différentes du modèle ResNet avec 18, 34, 50, 152 couches de convolutions. Pour chaque version du modèle avec chacune des images nous avons ensuite effectué 100 exécutions. Les observations et interprétations faites après les expérimentations sont les suivantes :

Observation 1 : <u>ResNet18 a les coûts en temps de traitement le plus bas</u>
L'histogramme à la figure 11 présente le temps moyen nécessaire à chaque modèle pour prédire la classe de chacune des images utilisées. On note que ResNet18 a le temps de traitement le plus bas quelque soit la taille de l'image passée en entrée.

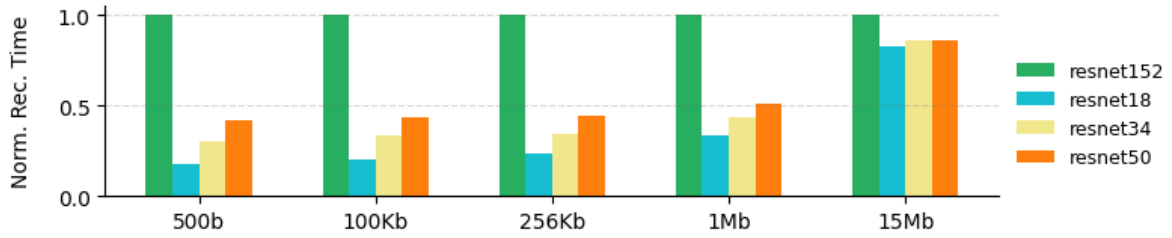

fig. 11: Temps moyen de prédiction

Observation 2 : <u>Resnet18 conserve les coûts en temps total le plus bas</u>

L'histogramme à la figure 12 présente les coûts moyens totaux en temps pour l'exécution du benchmark, incluant le temps pour prédire la classe de l'image, et le temps de téléchargement de l'image. On constate que ResNet18 conserve globalement un coût total en temps légèrement plus bas que les autres.

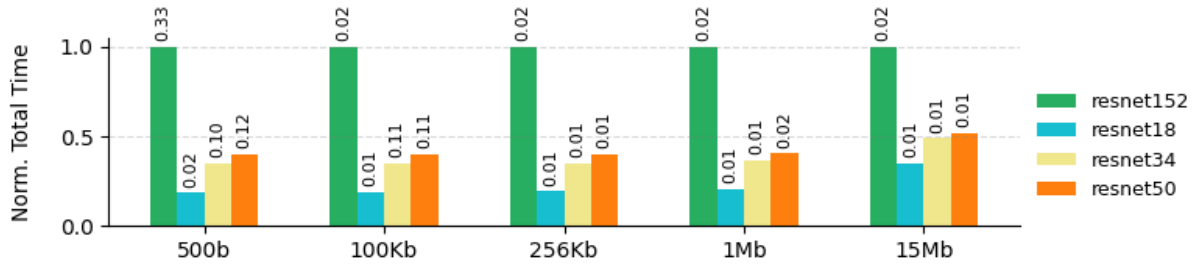

fig. 12: Temps moyen de total

Observation 3 : <u>Resnet18 a la consommation énergétique la plus basse</u>

L'histogramme à la figure 13 présente l'énergie moyenne consommée par chaque modèle avec des images de différentes tailles en entrée. On note principalement que ResNet18 a la consommation énergétique la plus basse pour toutes les images passées en entrée.

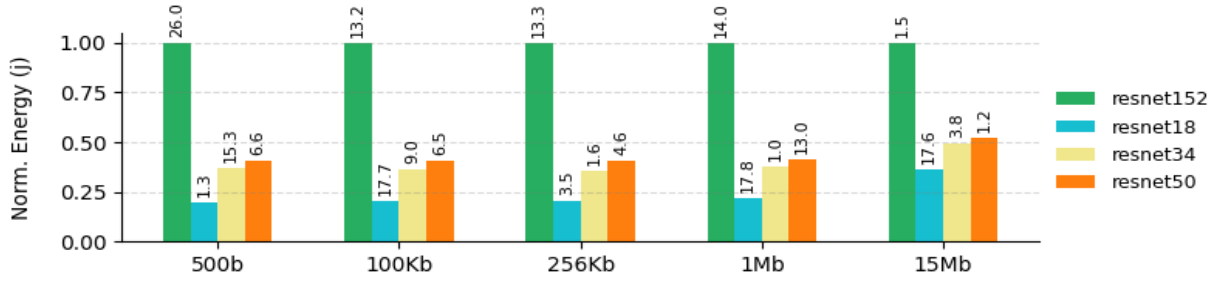

fig. 13: Consommation énergétique moyenne

Observation 4 : <u>Resnet152 et ResNet50 ont les prédictions les plus certaines</u>

Nous nous sommes ensuite intéressés aux performances de chaque variante du modèle ResNet en comparant la qualité de prédiction par chacune d'elle. L'histogramme à la figure 14 présente les probabilités en sortie pour chaque modèle représentant le degré de certitude du modèle quant à la prédiction faite pour l'image en entrée. Au sommet de chaque barre de l'histogramme est marqué la valeur de la probabilité. On constate que dans l'ensemble,

ResNet50 et ResNet152 sont plus performant car ils ont des probabilités les plus élevées, et donc un niveau de confiance plus grand.

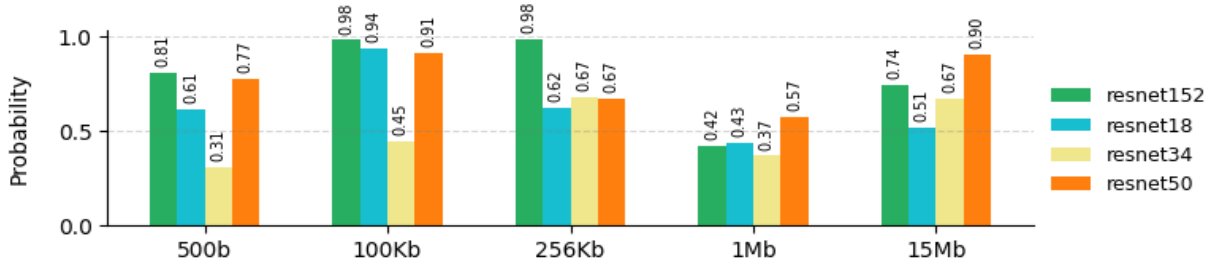

fig. 14: Probabilité en sortie

Interprétation côté utilisateur : En considérant les observations précédentes, et en effectuant une analyse du point de vue utilisateur, la meilleure alternative pour ce benchmark serait d'utiliser le modèle ResNet50, car avec ResNet152 ils sont plus performant mais ResNet18 a un coût en temps plus bas.

Interprétation côté fournisseur : Pour le fournisseur, le meilleur choix serait d'utiliser ResNet18 car il a des coûts en temps les plus bas, ainsi que des consommations énergétiques les plus basses.

Concernant la motivation : En ce qui concerne les objectifs du projet GreenFaaS, on constate là encore qu'il existe pour une tâche précise des implémentations alternatives avec des coûts en temps différents, des consommations énergétiques différentes, et des résultats différents mais acceptables satisfaisant séparément le fournisseur et l'utilisateur.

D. Text To Speech

Le benchmark met en œuvre une application qui convertit un fichier texte en audio. Elle reçoit un texte en entrée et produit en sortie un fichier audio encodé dans un format particulier. Le déroulement de l'exécution de l'application est illustré à la figure 1. Plusieurs étapes doivent donc être réalisées, à savoir l'étape de :

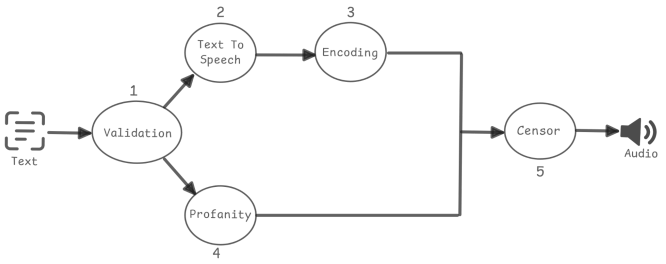

fig. 1: Text To Speech application workflow

1. **Validation:** l'objectif de cette étape est de s'assurer que le texte ne contient aucun élément pouvant poser des problèmes juridiques, tels que des discours haineux, des

menaces ou tout autre contenu illégal. Ne pouvant pas explicitement implémenter cette étape, nous l'avons simulé en exécutant simplement une boucle.

2. **Text To Speech**: Lors de cette étape, on transcrit le fichier texte en audio au format MP3 à l'aide des outils *espeak-ng* et *ffmpeg* disponible sur ubuntu.

3. **Encoding**: on convertit le fichier du format MP3 au format WAV. Cette conversion entraîne généralement une augmentation de la taille du fichier, mais de nombreuses applications nécessitent en entrée des fichiers au format WAV. La conversion est effectuée à l'aide de la bibliothèque *ffmpeg*.

4. **Profanity**: Elle se déroule parallèlement aux étapes précédentes. L'objectif est de détecter des insultes dans le texte, ou tout autre langage jugé inapproprié. Contrairement à la phase de validation, cette étape traite le texte d'un point de vue éthique plutôt que juridique. L'implémentation est effectuée avec la bibliothèque Python *profanity* , qui utilise un filtre basé sur une liste noire de mots.

5. **Censor**: Censure les parties d'un fichier audio WAV en fonction d'une liste d'expressions à censurer. Pour effectuer la censure, toutes les parties concernées sont réduites au silence à l'aide de la bibliothèque Python *wave*.

Les différents schéma d'exécution alternatifs que nous proposons pour cette application sont les suivants:

i. Exécuter uniquement l'étape 2 (Text To Speech). Pour cette alternative, on suppose que l'utilisateur ne va ni valider le texte, ni censurer l'audio, et accepte le format MP3 en sortie.

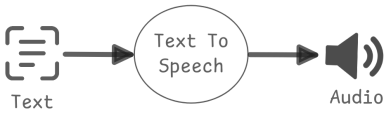

fig. 2: schema S1

ii. Exécuter l'étape 2 (Text To Speech) et ensuite l'étape 3 (encoding). Pour cette alternative, l'utilisateur ne valide pas le texte en entré, et aura en sortie un fichier audio non censuré mais encodé au format WAV.

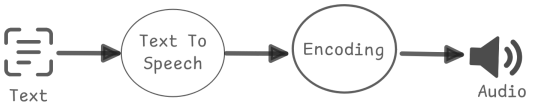

fig. 3: schema S2

iii. Exécuter toutes les étapes sauf l'étape de validation . Le texte en entrée ne sera pas validé, mais l'utilisateur aura en sortie un fichier audio censuré, et converti au format WAV.

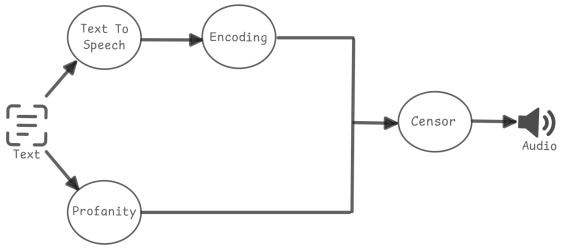

fig. 4: schema S3

iv. Exécuter l'étape 1 (validation) , l'étape 2 (Text To Speech) , et l'étape 3 (encoding). Pour cette alternative, le texte en entrée passe l'étape de validation, avant d'être transcrit en audio et convertit au format WAV, mais sans être censuré.

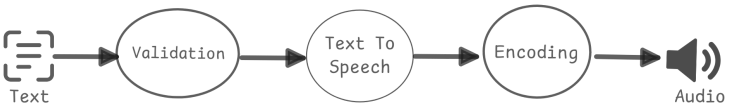

fig. 5: schema S4

iv. Pour le schéma S5, il s'agit d'exécuter l'ensemble des étapes du benchmark comme présenté précédemment à la figure 1.

Nous avons utilisé trois entrées différentes **: un fichier texte de 247 mots pesant 1 Kb, un autre de 791 mots pesant 5 Kb, et enfin un fichier de 2096 mots pesant 12 Kb**. Pour chaque schéma précédemment présenté avec chaque entrée, nous avons effectué 30 exécutions et les métriques collectés ont ensuite été normalisées de tel sorte que les valeurs les plus petites soient plus proches de 0 et les plus grandes plus proches de 1. Les observations et interprétations faites après les expérimentations sont les suivantes :

Observation 1 : <u>Le schéma d'exécution S1 à le temps de traitement le plus bas</u>
L'histogramme à la figure 6 permet de comparer le temps moyen de traitement de l'ensemble des fonctions pour chaque schéma. On constate que le schéma d'exécution S1 à le temps de traitement le plus bas pour toutes les entrées.

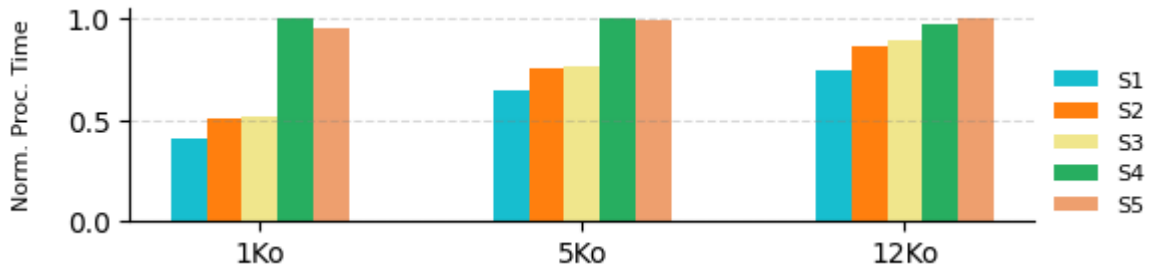

fig. 6: Average normalized Processing Time

Observation 2 : <u>Le schéma d'exécution S1 conservent le temps total le plus bas</u>

L'histogramme à la figure 7 présente les coûts moyens totaux en temps (temps de traitement plus les temps d'accès au stockage distant). On note principalement que S1 conserve le temps moyen total le plus bas.

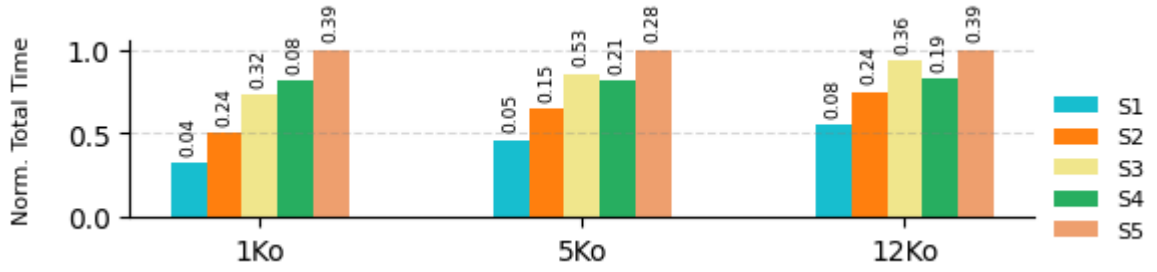

fig. 7: Average normalized total time

Observation 3 : <u>Le schéma d'exécution S1 à la consommation énergétique moyenne la plus petite, et S5 la grande</u>

Nous nous sommes ensuite intéressés à l'énergie consommée pendant l'exécution du benchmark. L'histogramme à la figure 8 présente l'énergie moyenne consommée lors des expérimentations pour chaque schéma d'exécution. On peut noter une corrélation entre l'énergie consommée et le temps d'exécution précédemment présenté, le schéma d'exécution S1 a donc une consommation énergétique plus basse que les autres.

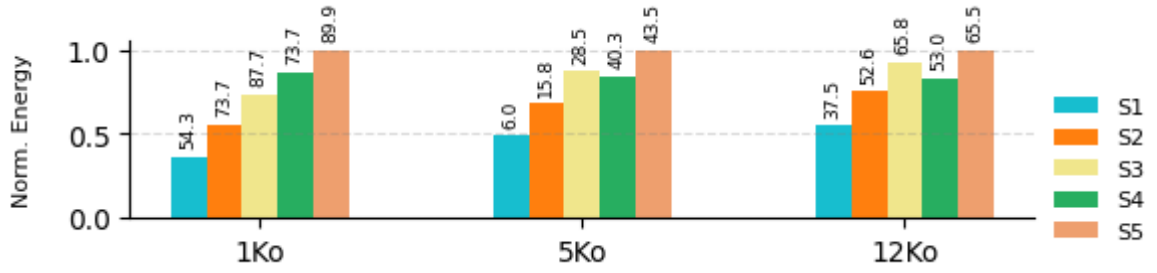

fig. 8: Average normalized energy

<u>Interprétation côté utilisateur</u> : En tenant compte des analyses effectuées précédemment, du point de vue de l'utilisateur on peut dire que en terme de fonctionnalité la meilleure alternative pour l'utilisateur serait le schéma d'exécution S5, car il fournit à l'utilisateur l'ensemble de fonctionnalité pour valider le texte en entré, et avoir en sortie un fichier audio censuré au format WAV.

<u>Interprétation côté fournisseur</u> : Pour le fournisseur, la meilleure alternative serait d'utiliser le schéma d'exécution S1 car il fournit des coûts en temps les plus bas, ainsi que la consommation énergétique la plus basse.

<u>Concernant la motivation, est-ce intéressant</u> ? En ce qui concerne les objectifs du projet GreenFaaS, pour ce benchmark, on constate qu'il existe effectivement des implémentations alternatives avec des coûts en temps différents, des consommations énergétiques différentes, et des résultats différents mais acceptables.

Reconnaissance Faciale

Le benchmark réalise une application de reconnaissance faciale. Elle prend en entrée une image contenant le visage d'un acteur et une vidéo, puis dessine un cadre autour du visage de l'acteur dans toutes les scènes de la vidéo. L'exécution de cette application se déroule parallèle, avec plusieurs unités de traitement qui exécutent simultanément six étapes, comme illustré à la figure 1 :

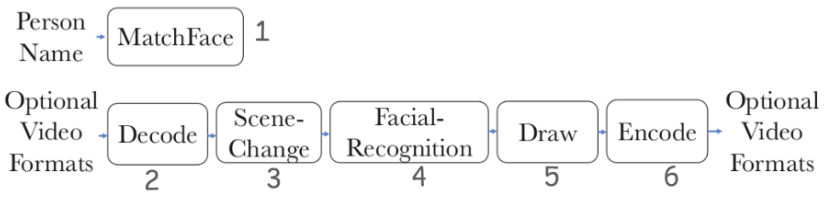

fig. 1: Image recognition application workflow

1. **MatchFace**: Cette étape consiste à trouver sur le web l'image de référence, celle qui contient déjà le visage de la personne cible et qu'on utilisera lors de l'étape reconnaissance faciale. L'exécution de cette étape est indépendante des autres.

2. **Le décodage**: Chaque unité de traitement récupère et décode une partie de la vidéo en blocs d'images de longueur fixe, puis stocke les résultats dans l'espace de stockage.

3. **Changement de scène**: Un algorithme de détection de changement de scène est ensuite appliqué à chaque le bloc d'image. L'objectif est de regrouper les images en scènes distinctes afin de faciliter les traitements à l'étape suivante.

4. **Reconnaissance faciale**: Après le changement de scène, chaque unité de traitement exécute un algorithme de reconnaissance faciale une fois toutes les n images d'une scène. Si au moins une image de la scène contient le visage cible, toutes les images de la scène sont marquées comme contenant le visage cible. L'algorithme renvoie aussi des informations indiquant la zone de l'image où se trouve le visage chercher.

5. **Le traçage** : Cette étape continue si les images d'une scène ont été étiquetées comme contenant le visage cible. Si c'est le cas, un cadre est dessiné à l'endroit indiqué à l'étape précédente, sinon rien ne se passe et les images constituant la scène sont conservées telles quelles.

6. **L'encodage:** Chaque unité de traitement encode (convertit) ensuite en un format d'encodage précis toutes les images ayant déjà été traitées pour reconstituer la partie de la vidéo.

Dans le cadre du projet GreenFaaS, les implémentations alternatives que nous proposons pour cette application sont les suivantes:

i. Ajoutez un chemin alternatif reliant l'étape 2 à l'étape 4' dans le graphe d'exécution. Comme illustré à la figure 4, après l'étape 2, il serait possible de passer directement à l'étape 4', où l'algorithme de reconnaissance faciale est appliqué à toutes les images constituant la vidéo. Contrairement à l'étape 4 qui nécessite préalablement de regrouper les images en scènes distinctes, pour ensuite appliquer l'algorithme à quelques images de chaque scène, l'étape 4 n'exige pas ce regroupement préalable en scènes.

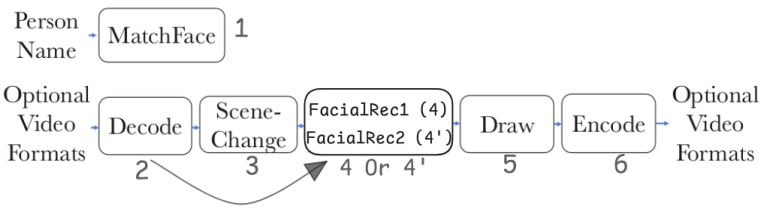

fig. 2: Image recognition application workflow first alternative

Avec cette alternative, l'algorithme de reconnaissance faciale devra être exécuté un plus grand nombre de fois. Cependant, cela permettra d'éviter l'étape 3 et d'identifier avec précision où se trouve le visage cible sur chaque image de la vidéo. En considérant cette approche et en se référant à la figure 2, il est possible de générer les schémas d'exécution suivants :

S1 : 2→3→4→5→6

S2 : 2→4'→5→6

ii. Ajoutez l'étape 5' au graphe d'exécution comme un chemin alternatif. Les auteurs suggèrent cette alternative où, plutôt que d'exécuter l'étape 5 qui consiste à dessiner un cadre autour du visage recherché, on pourrait introduire une étape 5' qui ne conserve que les scènes de la vidéo contenant le visage cible, et ignore les autres scènes.

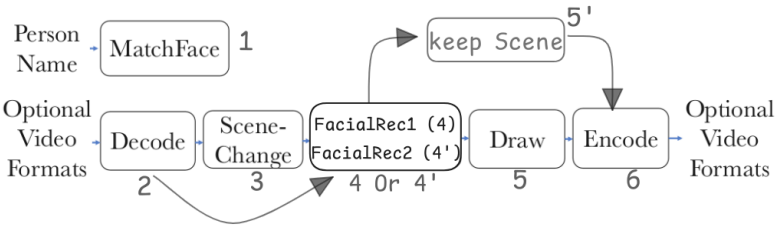

fig. 3: Image recognition application workflow second alternative

Comme précisé par les auteurs, cette alternative permettrait à l'utilisateur d'avoir en sortie une vidéo constituée uniquement avec des scènes contenant le visage cible. En observant la figure 3 on peut générer de nouveaux schémas d'exécution possible à savoir :

S3 : 2→3→4→5'→6

S4 : 2→4'→5'→6

iii. Ajoutez un chemin alternatif reliant l'étape 5' à l'étape 5 dans le graphe d'exécution. Pour fournir davantage de précision à l'utilisateur, il serait possible de conserver uniquement les scènes contenant le visage cible comme indiqué précédemment, mais aussi de dessiner un cadre indiquant l'emplacement exact du visage recherché dans chacune de ces scènes. Pour le faire, on doit exécuter l'étape 5 après avoir exécuté l'étape 5' comme indiqué à la figure 4.

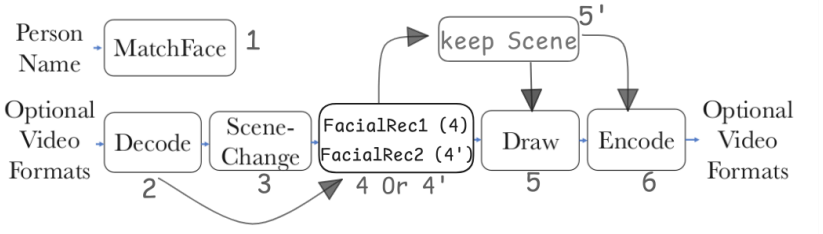

fig. 4: Image recognition application workflow third alternative

En prenant en compte cela, on obtient de nouveaux schéma d'exécution possible à savoir :

S5 : 2→3→4→5'→5→6

S6 : 2→4'→5'→5→6

La vidéo en entrée utilisée est une vidéo de résolution de **360p**, une durée totale de **35 secondes**. Pour chaque schéma précédemment présenté, nous avons effectué 30 exécutions. Chaque exécution correspond au traitement en parallèle de la vidéo par **10 processus lancés simultanément**. Chaque processus est donc chargé de traiter au moins 3 secondes de la vidéo en entrée. Après expérimentation, les résultats et observations que nous avons pu faire sont les suivantes :

Observation 1 : <u>Le schéma d'exécution S5 à le temps de traitement le plus bas</u>
L'histogramme à la figure 5 permet de comparer le temps moyen de traitement total (décodage, changement de scène, reconnaissance faciale, traçage, keep scène et encodage) de l'ensemble des 10 processus. On constate que le schéma d'exécution S3 à le temps de traitement le plus bas.

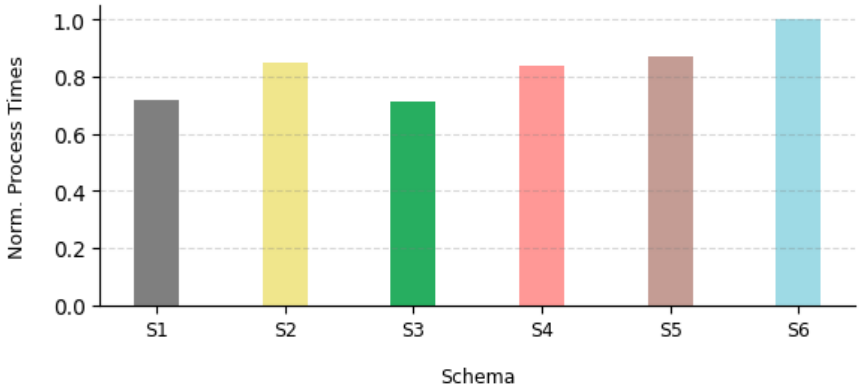

fig. 5: Average normalized Processing Time

Observation 2 : <u>Le schéma d'exécution S3 conservent le temps total le plus bas</u>
L'histogramme à la figure 6 présente les coûts moyens en temps total (temps de traitement et temps d'accès au stockage distant). On note principalement que S3 conserve le temps moyen total le plus bas.

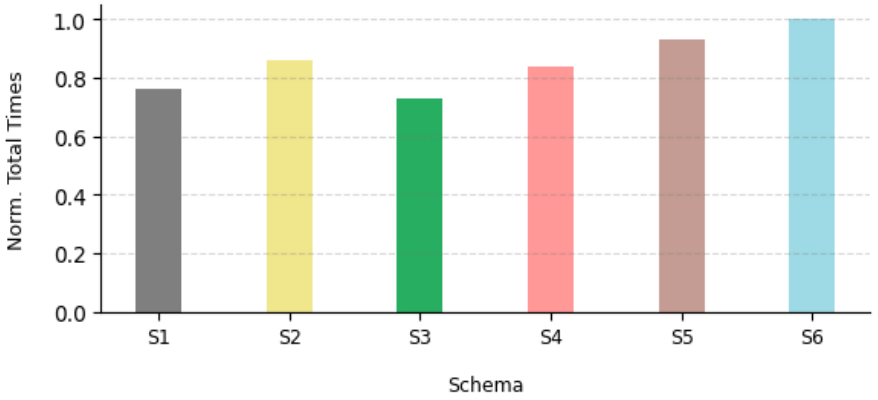

fig. 6: Average normalized time

Observation 3 : <u>Les schémas d'exécution S3 et S4 ont les consommations énergétiques moyennes les plus petites.</u>

Nous nous sommes ensuite intéressés à l'énergie consommée pendant l'exécution du benchmark. L'histogramme à la figure 7 présente l'énergie moyenne consommée lors des expérimentations avec chaque schéma d'exécution. On note principalement que le schéma d'exécution S5 a une consommation énergétique plus basse que les autres.

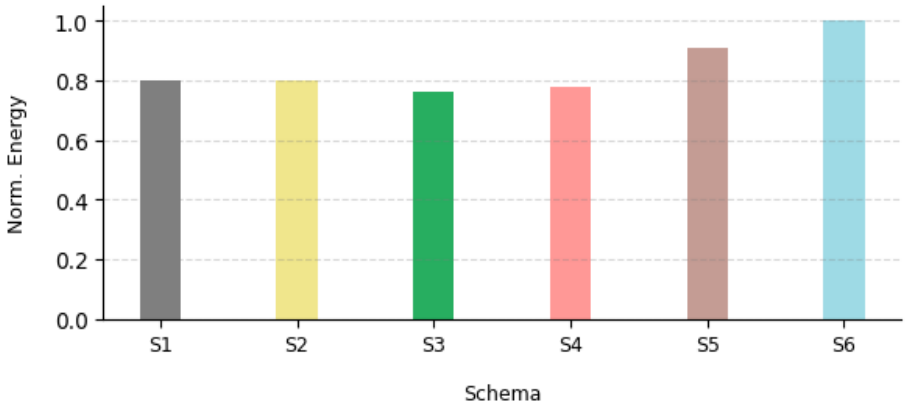

fig. 7: Average normalized energy

<u>Interprétation côté utilisateur</u> :

Pour l'utilisateur, l'alternative la plus adaptée pour ce benchmark serait le schéma d'exécution S2, car il fournit à l'utilisateur la possibilité de traiter et conserver la totalité des images constituant la vidéo en entrée.

<u>Interprétation côté fournisseur</u> :

Pour le fournisseur, la meilleure alternative serait d'utiliser le schéma d'exécution S3 car il fournit des coûts en temps les plus bas, ainsi que des consommations énergétiques les plus basses.