



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UE CALCUL INTENSIF : DISTRIBUTION DES DONNÉES ET
DES CALCULS

RabbitMQ : img-proc-mq

Etude de cas

MARTIN AVILA Charly

TAF ILSD
Année 2025

Table des matières

1	Introduction	2
2	Architecture logicielle	3
3	RabbitMQ	9
4	Conclusion et performances	12

1 Introduction

Dans un monde où les volumes de données augmentent de manière exponentielle, la gestion et le traitement efficace de ces données deviennent des enjeux majeurs. Le traitement d'images, en particulier, est un domaine clé dans de nombreuses applications, allant de l'imagerie médicale aux systèmes de reconnaissance visuelle. La complexité et la taille croissante des images nécessitent des solutions techniques capables de répondre aux exigences de rapidité, de scalabilité et de robustesse.

Le projet `img-prc-mq` s'inscrit dans cette optique en proposant un système distribué de traitement d'images, en particulier de la compression, basé sur RabbitMQ, une solution de messagerie légère et performante. L'objectif est de concevoir et d'implémenter une architecture logicielle modulaire, permettant d'uploader des images, de les compresser, puis de fournir les fichiers compressés aux utilisateurs finaux de manière efficace.

Le choix de RabbitMQ s'explique par ses nombreux avantages, notamment la capacité à gérer des flux massifs de messages, sa flexibilité dans la mise en œuvre de modèles de communication asynchrones et son support natif pour la persistance des messages. Cette approche garantit un système évolutif et résilient, adapté à des charges de travail variables.

L'architecture retenue repose sur des principes de découplage, de modularité et de scalabilité, en s'appuyant sur des microservices indépendants mais interconnectés. Chaque composant joue un rôle bien défini : Un serveur Flask gère les interactions avec les utilisateurs, des consumers RabbitMQ effectuent les tâches de traitement en arrière-plan, et RabbitMQ agit comme intermédiaire pour coordonner les flux de données entre les différents éléments.

Ce rapport détaille les étapes clés de la conception et de l'implémentation de ce système, en mettant en avant les choix techniques, les défis rencontrés et les solutions adoptées pour atteindre les objectifs du projet.

2 Architecture logicielle

L'architecture logicielle de ce projet repose sur une communication asynchrone entre plusieurs composants clés :

- **Client** : User interface pour lancer les requêtes.
- **Serveur Flask** : Gère les interactions avec les utilisateurs (upload d'images, suivi de statut, téléchargement des fichiers compressés).
- **Consumers RabbitMQ** : Exécutent les tâches de traitement des images (compression).
- **RabbitMQ** : Agit comme un système de messagerie, orchestrant les interactions entre le serveur et les consumers.

Cette architecture découplée permet une scalabilité et une résilience accrues, tout en assurant une maintenance simplifiée.

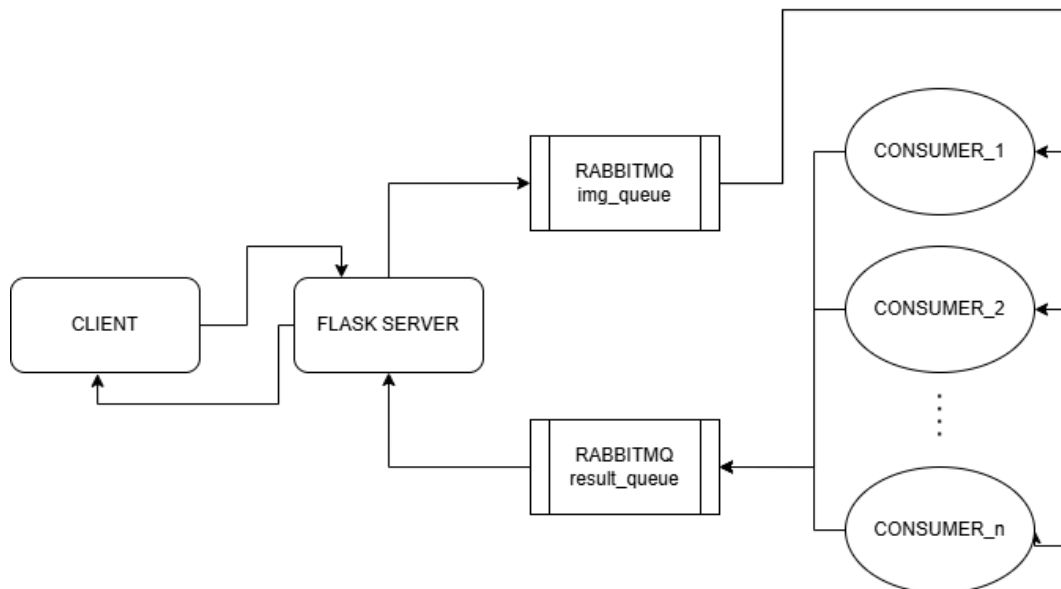


FIGURE 1 – Schéma de l'architecture logicielle

2.1 Serveur Flask

Le serveur Flask joue un rôle central dans l'architecture en tant que point d'entrée principal pour les utilisateurs et intermédiaire entre le client, RabbitMQ, et les consommateurs. Il est responsable des tâches suivantes :

- **Upload des images** : Le serveur reçoit les fichiers envoyés par les utilisateurs via une requête HTTP POST au point de terminaison `/upload`. Les images sont validées (type et extension), sauvegardées dans le dossier dédié `./uploads` (volume partagé), puis une tâche contenant le chemin du fichier et l'action à effectuer est publiée dans la file `TASK_QUEUE` de RabbitMQ.

Exemple de code pour l'upload et la publication d'une tâche dans RabbitMQ :

```

1 @app.route('/upload', methods=['POST'])
2 def upload_image():
3     if 'image' not in request.files:
4         return jsonify({"message": "No_file_part"}),
5             400
6
7     file = request.files['image']
8     if file.filename == '':
9         return jsonify({"message": "No_selected_file"
10             }), 400
11
12     if not allowed_file(file.filename):
13         return jsonify({"message": "File_type_not_
14             allowed"}), 400
15
16     file_path = save_file(file)
17     task_message = json.dumps({"file_path": file_path,
18         "action": "compress"})
19     publish_message(RABBITMQ_SERVER, TASK_QUEUE,
20         task_message)
21
22     return jsonify({"message": "File_uploaded_
23         successfully", "file_path": file_path}), 200

```

- **Suivi du statut** : Le point de terminaison `/status` permet aux utilisateurs de vérifier si leur image a été compressée. Si le traitement est terminé, un lien de téléchargement est généré dynamiquement.

Exemple de code pour vérifier le statut d'une image compressée :

```

1 @app.route('/status', methods=['GET'])
2 def check_status():
3     file_name = request.args.get('file_name')
4     if not file_name:
5         return jsonify({"message": "File_name_is_
6             missing"}), 400
7
8     compressed_path = os.path.join(app.config['
9         DOWNLOAD_FOLDER'], f"compressed_{file_name}")
10    if os.path.exists(compressed_path):
11        download_url = url_for('download_file',
12            file_name=f"compressed_{file_name}",
13            _external=True)
14        return jsonify({"status": "completed", "
15            download_url": download_url}), 200
16    else:
17        return jsonify({"status": "processing"}), 200

```

- **Téléchargement des images compressées** : Une fois la compression terminée, les utilisateurs peuvent récupérer leur fichier via le point de terminaison `/download`. Le serveur sert les fichiers compressés à partir du dossier `downloads/` en tant que pièces jointes prêtes à être téléchargées.

Exemple de code pour le téléchargement d'une image compressée :

```
1 @app.route('/download', methods=['GET'])
2 def download_file():
3     file_name = request.args.get('file_name')
4     if not file_name:
5         return jsonify({"message": "File_name_is_
6             missing"}), 400
7
8     file_path = os.path.join(app.config['
9         DOWNLOAD_FOLDER'], file_name)
10    if not os.path.exists(file_path):
11        return jsonify({"message": "File_not_found"}),
12        404
13
14    return send_file(file_path, as_attachment=True)
```

Principaux modules utilisés

Le serveur utilise des bibliothèques externes pour lancer le serveur, et la connexion à RabbitMQ

- **Flask** : Framework léger pour gérer les requêtes HTTP et servir les fichiers.
- **Flask-CORS** : Gère les problèmes liés aux requêtes inter-origines (*Cross-Origin Resource Sharing*).
- **pika** : Librairie Python pour interagir avec RabbitMQ.
- **Modules internes** :
 - `utils.rabbitmq` : Pour publier et consommer des messages RabbitMQ.
 - `utils.file_handler` : Pour valider et sauvegarder les fichiers localement.

Avantages

- **Modularité** : Une architecture découplée qui permet une maintenance simplifiée.
- **Simplicité d'utilisation** : Flask offre une interface intuitive pour gérer les requêtes utilisateur.
- **Scalabilité** : Facilité d'ajouter des fonctionnalités ou de supporter un plus grand nombre de requêtes.

Limites

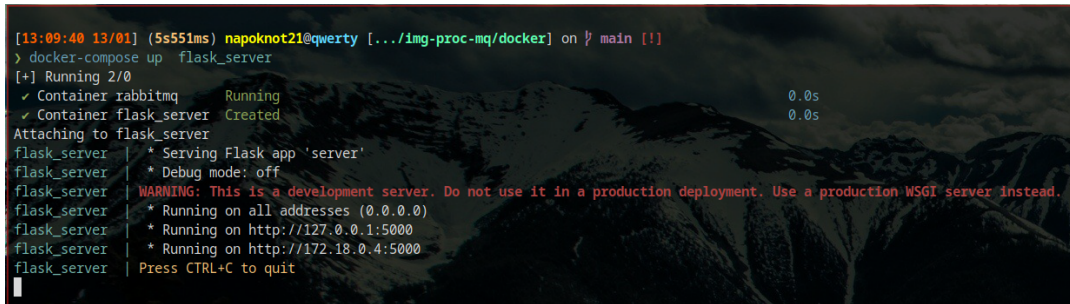
- **Performance** : Sous une charge très importante, un équilibrage de charge (load balancing) serait nécessaire.

- **Résilience** : Une gestion centralisée des logs et des erreurs pourrait être ajoutée pour améliorer la robustesse.

Build

Cette partie du projet peut se build (et lancer) via deux façons :

- **Docker** : Dans le dossier `docker/` l'image `Docker.server` permet de lancer dans un conteneur pour mieux simuler un environnement distribué.
- **Localhost** : Après certaines modifications expliquées dans le `README.md` du projet, on peut lancer localement le serveur.



```
[13:09:40 13/01] (5s551ms) napoknot21@qwertz [.../img-proc-mq/docker] on 1 main [!]  
> docker-compose up flask_server  
[+] Running 2/0  
✓ Container rabbitmq      Running      0.0s  
✓ Container flask_server  Created      0.0s  
Attaching to flask_server  
flask_server | * Serving Flask app 'server'  
flask_server | * Debug mode: off  
flask_server | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.  
flask_server | * Running on all addresses (0.0.0.0)  
flask_server | * Running on http://127.0.0.1:5000  
flask_server | * Running on http://172.18.0.4:5000  
flask_server | Press CTRL+C to quit
```

FIGURE 2 – Lancement du serveur Flask

2.2 Consumers RabbitMQ

Les consumers RabbitMQ sont des composants essentiels de l'architecture. Ils exécutent les tâches de traitement des images (par exemple, la compression) qui leur sont assignées via RabbitMQ. Chaque consumer est indépendant, ce qui permet une scalabilité horizontale et une répartition efficace des charges.

- **Réception des tâches** : Les consumers récupèrent les messages publiés dans la file `TASK_QUEUE` par le serveur Flask.
- **Traitement des images** : Après avoir reçu une tâche, le consumer compresse l'image spécifiée en utilisant la bibliothèque `Pillow`.
- **Publication des résultats** : Les résultats (chemin de l'image compressée) sont publiés dans la file `RESULT_QUEUE`, prêts à être consommés par le serveur Flask.

Fonctionnement général

Le fonctionnement des consumers suit un workflow en trois étapes :

1. Connexion à RabbitMQ et initialisation de la file `TASK_QUEUE`.
2. Consommation des messages contenant les informations sur les tâches.
3. Compression des images et publication des résultats dans `RESULT_QUEUE`.

Exemple de code pour la configuration et le démarrage d'un consumer :

```

1 from rabbitmq.connection import get_channel
2 from core.compressor_consumer import process_message
3 from settings.config import RABBITMQ_SERVER, TASK_QUEUE
4
5 def main():
6     connection, channel = get_channel(RABBITMQ_SERVER)
7     channel.queue_declare(queue=TASK_QUEUE, durable=True)
8
9     print(f"[*] ␣Waiting␣for␣messages␣in␣{TASK_QUEUE}...")
10
11     channel.basic_qos(prefetch_count=1)
12     channel.basic_consume(queue=TASK_QUEUE,
13                           on_message_callback=process_message)
14
15     try:
16         channel.start_consuming()
17     except KeyboardInterrupt:
18         print("\n[-] ␣Consumer␣stopped␣manually.␣Bye!")
19     finally:
20         connection.close()
21
22 if __name__ == "__main__":
23     main()

```

Compression des images

La compression est effectuée en utilisant la bibliothèque Pillow.

Chaque image est convertie en format JPEG avec une qualité spécifiée pour réduire sa taille.

Exemple de code pour la compression d'une image :

```

1 import os
2 from PIL import Image
3
4 def compress_image (file_path) :
5     try:
6         with Image.open(file_path) as img:
7             compressed_path = ROOT_STORAGE_FOLDER +
8                               DOWNLOAD_FOLDER + f"compressed_{os.path.
9                               basename(file_path)}"
10             img = img.convert("RGB")
11             img.save(compressed_path, "JPEG", optimize=
12                      True, quality=85)
13             print(f"\n[+] ␣Image␣compressed:␣{
14                   compressed_path}\n")
15             return compressed_path
16     except Exception as e:
17         raise RuntimeError(f"[-] ␣Failed␣to␣compress␣
18                            image:␣{e}")

```


Publication des résultats

Une fois la compression terminée, les consumers publient les informations sur l'image compressée dans la file `RESULT_QUEUE`, permettant au serveur Flask de finaliser le workflow.

Exemple de code pour la publication des résultats :

```
1 from rabbitmq.publisher import publish_message
2 from settings.config import RESULT_QUEUE
3
4 def process_message(ch, method, properties, body):
5     message = json.loads(body)
6     file_path = message.get("file_path")
7
8     try:
9         compressed_path = compress_image(file_path)
10        result_message = {"original_path": file_path, "
11                           compressed_path": compressed_path}
12        publish_message(RESULT_QUEUE, result_message)
13        ch.basic_ack(delivery_tag=method.delivery_tag)
14    except Exception as e:
15        ch.basic_nack(delivery_tag=method.delivery_tag)
```

Principaux modules utilisés

- **Pillow** : Pour effectuer les opérations de traitement et compression d'images.
- **pika** : Pour interagir avec RabbitMQ (consommation et publication des messages).
- **Modules internes** :
 - `core.compressor_consumer` : Pour le traitement des messages et la compression des images.
 - `rabbitmq.connection` : Pour établir une connexion avec RabbitMQ.
 - `rabbitmq.publisher` : Pour publier les résultats dans la file `RESULT_QUEUE`.

Avantages

- **Scalabilité horizontale** : Possibilité d'ajouter plusieurs instances de consumers pour traiter un grand volume de messages.
- **Indépendance** : Les consumers fonctionnent indépendamment du serveur Flask, permettant une meilleure modularité.
- **Résilience** : RabbitMQ garantit que les messages non traités restent disponibles jusqu'à leur consommation.

Limites

- **Performance limitée par le traitement** : Les tâches complexes comme la compression d'images volumineuses peuvent ralentir les performances.
- **Dépendance à RabbitMQ** : Si RabbitMQ devient indisponible, le traitement des tâches est interrompu.

Build

Les consumers peuvent être exécutés dans différents environnements :

- **Docker** : L'image Docker dédiée (`Dockerfile.consumer`) facilite le déploiement et l'exécution dans des conteneurs isolés.
- **Localhost** : En configurant les dépendances localement, il est possible de démarrer les consumers en local pour des tests rapides.

```
[13:09:47 13/01] napoknot21@qwerty [.../img-proc-mq/docker] on 1 main [!]  
> docker-compose up compression_consumer_1  
[+] Running 2/0  
✓ Container rabbitmq Running 0.0s  
✓ Container compression_consumer_1 Created 0.0s  
Attaching to compression_consumer_1
```

FIGURE 3 – Lancement du consumer

3 RabbitMQ

RabbitMQ est au cœur de l'architecture logicielle, agissant comme un système de messagerie fiable et performant. Il facilite la communication asynchrone entre les différents composants, assurant le découplage et la scalabilité de l'ensemble du système.

Rôles principaux

- **Médiateur de messages** : RabbitMQ reçoit, stocke et distribue les messages entre le serveur Flask et les consumers.
- **Persistant et fiable** : Les messages dans les queues sont marqués comme durables, garantissant qu'ils ne sont pas perdus en cas de panne.
- **Orchestration des tâches** : Permet de gérer plusieurs instances de consumers en répartissant les messages selon le principe du round-robin.

Files utilisées et exemples de messages

- **TASK_QUEUE** :
Contient les messages envoyés par le serveur Flask pour les tâches à traiter. Chaque message inclut les informations suivantes :
 - `file_path` : Le chemin de l'image à traiter.
 - `action` : L'action à effectuer (par exemple, `compress`).

Exemple de message envoyé par le serveur Flask :

```
1 {  
2     "file_path": "../uploads/user_image.jpg",  
3     "action": "compress"  
4 }
```

- **RESULT_QUEUE** :
Contient les résultats générés par les consumers. Chaque message inclut :
 - `original_path` : Le chemin de l'image originale.

- `compressed_path` : Le chemin du fichier compressé.

Exemple de message envoyé par un consumer :

```

1 {
2   "original_path": "./uploads/user_image.jpg",
3   "compressed_path": "./downloads/compressed_user_image.
4   jpg"
}
```

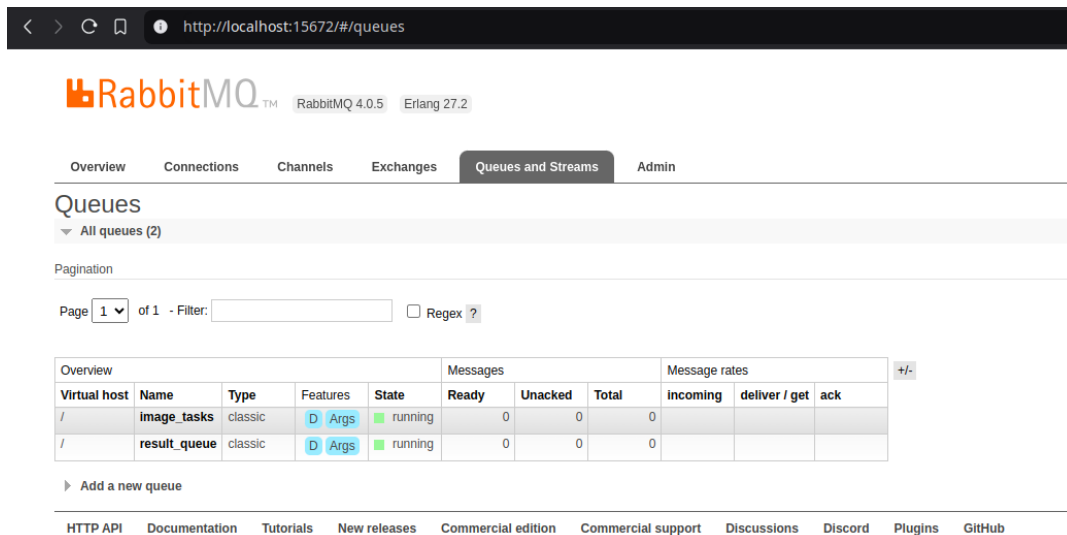


FIGURE 4 – Queues créées sur RabbitMQ

Configuration

RabbitMQ est configuré pour assurer une communication fiable et performante :

- Les queues (`TASK_QUEUE` et `RESULT_QUEUE`) sont déclarées comme **durables**, assurant la persistance des messages.
- Les **préférences de qualité de service** (QoS) sont configurées avec `prefetch_count=1`, permettant à chaque consumer de traiter un message à la fois pour éviter les surcharges.
- L'authentification par défaut (`guest:guest`) est utilisée dans un environnement de développement, mais elle peut être personnalisée pour une utilisation en production.

Avantages de RabbitMQ

- **Découplage des composants** : Les services client, serveur et consumer fonctionnent indépendamment, ce qui simplifie leur maintenance et leur évolution.
- **Fiabilité** : Les messages non traités restent disponibles jusqu'à leur consommation, garantissant qu'aucune tâche n'est perdue.
- **Scalabilité horizontale** : RabbitMQ permet d'ajouter ou de retirer dynamiquement des consumers selon la charge du système.

Limites

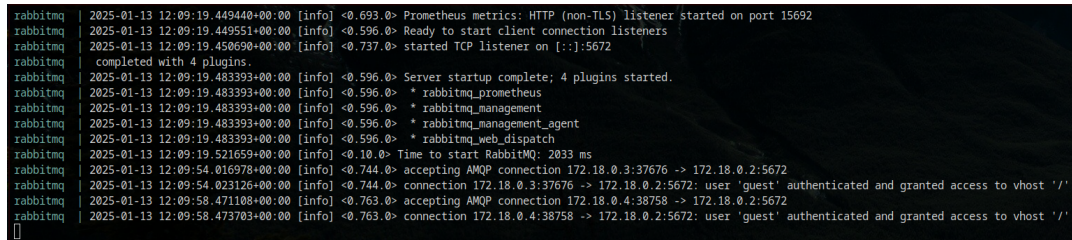
- **Dépendance réseau** : En cas de défaillance de la connexion réseau entre le serveur Flask et RabbitMQ, les messages ne peuvent pas être publiés ou consommés.
- **Complexité en production** : Une configuration avancée est nécessaire pour garantir la sécurité, l'équilibrage de charge et la gestion des échecs.

Build

Dans le projet, RabbitMQ est lancé en tant que conteneur Docker via le fichier `docker-compose.yml`. Voici les principales configurations :

- **Port 5672** : Utilisé pour les protocoles AMQP (Advanced Message Queuing Protocol).
- **Port 15672** : Utilisé pour l'interface de gestion Web.
- **Exemple de commande pour accéder à RabbitMQ via Docker** :

```
1 docker-compose up rabbitmq
```



```
rabbitmq | 2025-01-13 12:09:19.449440+00:00 [info] <0.693.0> Prometheus metrics: HTTP (non-TLS) listener started on port 15692
rabbitmq | 2025-01-13 12:09:19.449551+00:00 [info] <0.596.0> Ready to start client connection listeners
rabbitmq | 2025-01-13 12:09:19.450690+00:00 [info] <0.737.0> started TCP listener on [::]:5672
rabbitmq | completed with 4 plugins.
rabbitmq | 2025-01-13 12:09:19.483393+00:00 [info] <0.596.0> Server startup complete; 4 plugins started.
rabbitmq | 2025-01-13 12:09:19.483393+00:00 [info] <0.596.0> * rabbitmq_prometheus
rabbitmq | 2025-01-13 12:09:19.483393+00:00 [info] <0.596.0> * rabbitmq_management
rabbitmq | 2025-01-13 12:09:19.483393+00:00 [info] <0.596.0> * rabbitmq_management_agent
rabbitmq | 2025-01-13 12:09:19.483393+00:00 [info] <0.596.0> * rabbitmq_web_dispatch
rabbitmq | 2025-01-13 12:09:19.521659+00:00 [info] <0.10.0> Time to start RabbitMQ: 2033 ms
rabbitmq | 2025-01-13 12:09:54.016978+00:00 [info] <0.744.0> accepting AMQP connection 172.18.0.3:37676 -> 172.18.0.2:5672
rabbitmq | 2025-01-13 12:09:54.023126+00:00 [info] <0.744.0> connection 172.18.0.3:37676 -> 172.18.0.2:5672: user 'guest' authenticated and granted access to vhost '/'
rabbitmq | 2025-01-13 12:09:58.471108+00:00 [info] <0.763.0> accepting AMQP connection 172.18.0.4:38758 -> 172.18.0.2:5672
rabbitmq | 2025-01-13 12:09:58.473703+00:00 [info] <0.763.0> connection 172.18.0.4:38758 -> 172.18.0.2:5672: user 'guest' authenticated and granted access to vhost '/'
[]
```

FIGURE 5 – Lancement de RabbitMQ sur Docker

4 Conclusion et performances

4.1 Résumé du projet

Ce projet a mis en œuvre une architecture logicielle basée sur RabbitMQ pour orchestrer un système distribué de traitement d'images. Le découplage des responsabilités entre les différents composants (serveur Flask, consumers RabbitMQ et client) a permis de créer une solution robuste et évolutive. La modularité de cette approche simplifie la maintenance et offre la possibilité d'étendre facilement les fonctionnalités.

4.2 Performances du système

L'architecture a démontré une réactivité satisfaisante dans les cas de test. Voici les principaux résultats observés :

- **Temps de traitement moyen** : Une image de 3 MB est compressée en environ 2 secondes, incluant le temps de transfert des messages dans RabbitMQ.
- **Réduction de taille des images** : En moyenne, les images compressées atteignent 40% de leur taille initiale, tout en conservant une qualité visuelle acceptable grâce à des réglages optimaux (qualité à 85%).
- **Gestion de charge** : Avec l'ajout de plusieurs consumers, le système distribue efficacement les tâches, réduisant ainsi le temps d'attente global en cas de forte affluence.
- **Résilience** : Les tâches sont correctement conservées dans RabbitMQ en cas de panne d'un consumer, garantissant qu'aucune demande n'est perdue.

4.3 Perspectives d'amélioration

Bien que le système ait montré de bonnes performances, des axes d'amélioration ont été identifiés :

- **Optimisation des performances** : Intégrer un mécanisme de détection pour éviter la recompression inutile des images déjà optimisées.
- **Suivi en temps réel** : Ajouter un retour visuel pour les utilisateurs, avec des indicateurs de progression des tâches.
- **Extension des fonctionnalités** : Envisager l'ajout d'autres types de traitement, comme le redimensionnement ou le filtrage des images.
- **Tests à grande échelle** : Simuler des milliers de requêtes pour évaluer la scalabilité dans des environnements de production.

4.4 Conclusion finale

En conclusion, ce projet a permis de démontrer les avantages d'une architecture distribuée pour le traitement d'images à grande échelle. L'utilisation de RabbitMQ a été déterminante pour assurer la flexibilité, la fiabilité et la scalabilité du système. Ce cas d'étude constitue une base solide pour des applications plus complexes nécessitant une orchestration distribuée, tout en offrant des performances adaptées à des scénarios réels.