

---

# OpenFaaS sur Kubernetes : Fonctions Serverless de A à Z

Guide complet pour installer, configurer et maîtriser OpenFaaS sur Kubernetes : développement de fonctions serverless, déploiement, scaling automatique, surveillance, intégration avec des services externes et bonnes pratiques.

**DevOps** **Kubernetes** **60 min de lecture** **Niveau Intermédiaire**

---

Document généré le 13/05/2026 à 11h11 · [novy.fr/wiki/openfaas-kubernetes-serverless](https://novy.fr/wiki/openfaas-kubernetes-serverless)

# Sommaire

9 section(s) · 60 min de lecture

## Compétences à acquérir

- ↳ 1. Introduction à OpenFaaS
- ↳ 2. Préparation de l'environnement Kubernetes
- ↳ 3. Installation d'OpenFaaS sur Kubernetes
- ↳ 4. Exploration de l'interface utilisateur OpenFaaS
- ↳ 5. Création de fonctions
- ↳ 6. Développement d'une fonction serverless
- ↳ 7. Empaquetage et déploiement de fonctions
- ↳ 8. Test de la fonction

# Compétences à acquérir

---

À l'issue de ce guide, vous serez capable de :

- **Installer et configurer** OpenFaaS sur un cluster Kubernetes
  - **Développer et déployer** des fonctions serverless avec OpenFaaS
  - **Gérer et surveiller** les fonctions, logs et performances
  - **Utiliser le CLI et l'UI** d'OpenFaaS pour le déploiement et la gestion
  - **Intégrer OpenFaaS** avec d'autres services pour créer des applications serverless complètes
- 

## 1. Introduction à OpenFaaS

### Qu'est-ce qu'OpenFaaS ?

**OpenFaaS** (Open Functions as a Service) est une plateforme open source qui permet de transformer n'importe quel processus ou conteneur Docker en une **fonction serverless**. Elle fonctionne nativement sur **Kubernetes** et **Docker Swarm**.

▢ **L'idée centrale** : Tu écris une fonction, OpenFaaS s'occupe de tout le reste — packaging, déploiement, scaling, monitoring.

### Pourquoi le Serverless ?

Sans Serverless	Avec Serverless (OpenFaaS)
Tu gères les serveurs	Tu gères uniquement ton code
Scaling manuel	Scaling automatique
Payer même quand rien ne tourne	Scale à zéro possible
Configuration complexe	Déploiement en une commande
Maintenance lourde	Maintenance minimale

### Avantages d'OpenFaaS

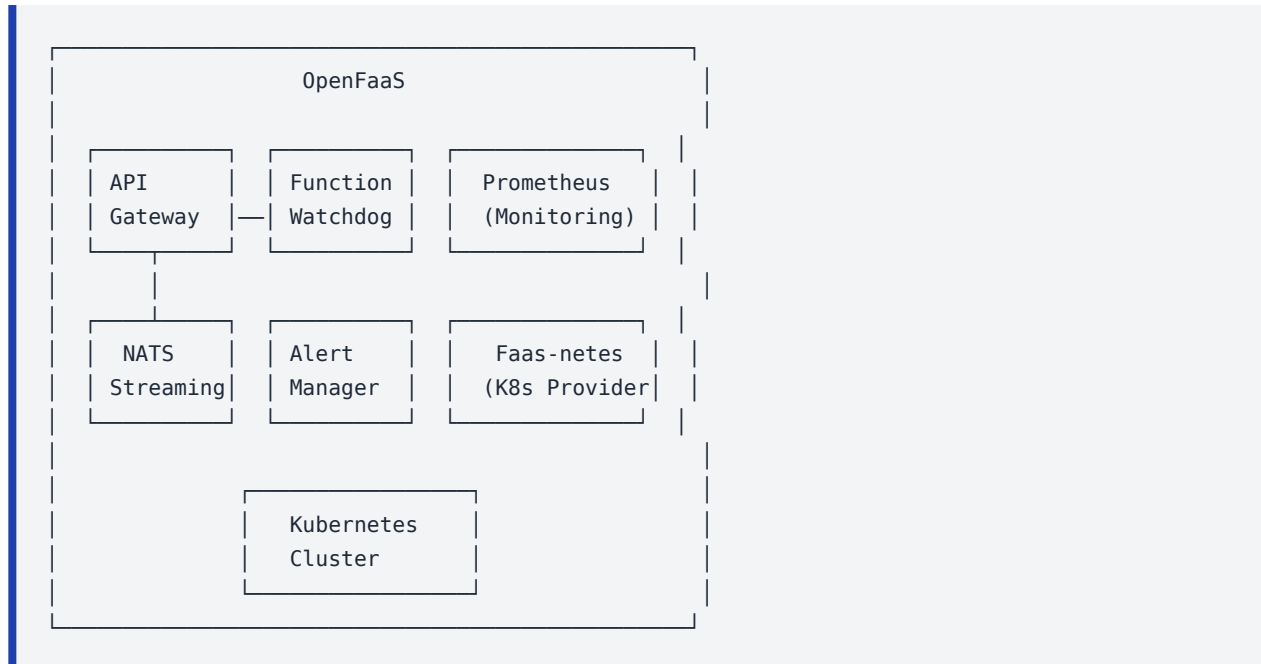
- **Open Source** : Pas de vendor lock-in (contrairement à AWS Lambda, Azure Functions)
- **Multi-langage** : Python, Node.js, Go, Java, C#, PHP, Ruby, et même des binaires
- **Portable** : Fonctionne sur n'importe quel Kubernetes (on-premise sur Debian comme ici, cloud, edge)
- **Simple** : Une fonction = un conteneur Docker
- **Extensible** : Templates personnalisables, triggers asynchrones, chaînes de fonctions

### Cas d'utilisation typiques

- **Traitement d'images** : Redimensionnement, compression, conversion
- **Envoi d'emails et notifications** : Déclenché par des événements
- **API Gateway** : Endpoints REST légers

- **Traitement de données** : ETL, transformations, agrégations
- **Webhooks** : Réception et traitement d'événements externes
- **Traitement de texte** : Parsing, analyse, génération

## Architecture d'OpenFaaS



📄 Copier

- **API Gateway** : Point d'entrée unique pour toutes les fonctions. Gère le routage, l'authentification et le scaling.
- **Function Watchdog** : Agent intégré dans chaque conteneur de fonction. Convertit les requêtes HTTP en appels de fonction.
- **faas-netes** : Provider Kubernetes qui gère les déploiements, services et réplicas.
- **NATS Streaming** : File de messages pour les invocations asynchrones.
- **Prometheus** : Collecte les métriques pour le monitoring et l'auto-scaling.
- **Alert Manager** : Déclenche le scaling basé sur les alertes Prometheus.

## 2. Préparation de l'environnement Kubernetes

### Révision des concepts Kubernetes essentiels

Avant d'installer OpenFaaS, assurez-vous de maîtriser ces concepts :

Concept	Description	Analogie
<b>Pod</b>	Plus petite unité déployable, contient un ou plusieurs conteneurs	Un appartement
<b>Service</b>	Expose un pod ou un ensemble de pods sur le réseau	L'adresse postale de l'appartement
<b>Deployment</b>	Gère le cycle de vie des pods (réplicas, mises à jour)	Le syndic de l'immeuble
<b>Namespace</b>	Isolation logique des ressources dans le cluster	Un étage de l'immeuble
<b>ConfigMap</b>	Stocke la configuration non sensible	Le tableau d'affichage
<b>Secret</b>	Stocke les données sensibles (mots de passe, tokens)	Le coffre-fort

## Prérequis

- **Un cluster Kubernetes on-premise fonctionnel** (1 master + 2 workers sous Debian)
- **kubectl** installé et configuré sur le master (ou votre poste de travail)
- **Helm 3** installé (gestionnaire de paquets Kubernetes)
- **Docker** installé sur chaque nœud du cluster (pour construire et exécuter les images)

## Vérifier votre environnement

```
# Vérifier kubectl
kubectl version --client

# Vérifier la connexion au cluster
kubectl cluster-info

# Vérifier les nœuds (vous devez voir 1 master + 2 workers)
kubectl get nodes
# Résultat attendu :
# NAME          STATUS    ROLES          AGE   VERSION
# master        Ready    control-plane  XXd   v1.XX.X
# worker-1      Ready    <none>         XXd   v1.XX.X
# worker-2      Ready    <none>         XXd   v1.XX.X

# Vérifier Helm
helm version
```

📄 Copier

## Installer Helm 3 sur Debian (si nécessaire)

```
# Télécharger et installer Helm
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

# Vérifier l'installation
helm version
```

📄 Copier

**i Cluster on-premise Debian** : Ce guide est conçu pour un cluster Kubernetes on-premise composé d'un **master** et de **2 workers** sous Debian. Les commandes `kubectl` sont exécutées depuis le nœud master ou depuis un poste de travail configuré avec le fichier `kubeconfig` approprié.

---

## 3. Installation d'OpenFaaS sur Kubernetes

### Étape 1 : Créer les namespaces

OpenFaaS utilise deux namespaces :

- `openfaas` : pour les composants du système (gateway, prometheus, etc.)
- `openfaas-fn` : pour les fonctions déployées

```
kubectl apply -f https://raw.githubusercontent.com/openfaas/faas-netes/master/namespaces.yml
```

📄 Copier

Ou manuellement :

```
kubectl create namespace openfaas  
kubectl create namespace openfaas-fn
```

📄 Copier

## Étape 2 : Installer arkade et OpenFaaS

**arkade** est l'outil officiel recommandé par OpenFaaS. Il gère automatiquement les namespaces, les secrets, le chart Helm et tous les composants.

```
# Installer arkade  
curl -sLsf https://get.arkade.dev/ | sudo sh  
  
# Installer OpenFaaS CE (Community Edition)  
arkade install openfaas-ce  
  
# Récupérer le mot de passe généré automatiquement  
PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode; echo)  
echo $PASSWORD > ~/openfaas-password.txt  
echo "Mot de passe OpenFaaS : $PASSWORD"
```

📄 Copier

## Étape 3 : Vérifier l'installation

```
# Vérifier les pods OpenFaaS  
kubectl -n openfaas get pods  
  
# Résultat attendu :  
# NAME                                READY   STATUS    RESTARTS   AGE  
# alertmanager-xxxxx                 1/1     Running   0           1m  
# basic-auth-plugin-xxxxx             1/1     Running   0           1m  
# gateway-xxxxx                       2/2     Running   0           1m  
# nats-xxxxx                          1/1     Running   0           1m  
# prometheus-xxxxx                   1/1     Running   0           1m  
# queue-worker-xxxxx                 1/1     Running   0           1m  
  
# Vérifier les services  
kubectl -n openfaas get svc
```

📄 Copier

## Étape 4 : Installer le CLI OpenFaaS (faas-cli)

```
# Sur Debian/Linux (depuis le master ou votre poste de travail)  
curl -sL https://cli.openfaas.com | sudo sh  
  
# Vérifier l'installation  
faas-cli version
```

📄 Copier

## Étape 5 : Se connecter au gateway OpenFaaS

```
# Récupérer l'IP du NodePort du gateway
GATEWAY_PORT=$(kubectl -n openfaas get svc gateway-external -o
jsonpath='{.spec.ports[0].nodePort}')
MASTER_IP=$(kubectl get nodes -l node-role.kubernetes.io/control-plane -o
jsonpath='{.items[0].status.addresses[?(@.type=="InternalIP")].address}')

# L'URL du gateway est accessible via n'importe quel nœud du cluster (master ou workers)
GATEWAY_URL=http://$MASTER_IP:$GATEWAY_PORT
echo "Gateway URL : $GATEWAY_URL"

# Alternativement, utiliser port-forward depuis le master :
kubectl port-forward -n openfaas svc/gateway 8080:8080 --address 0.0.0.0 &
# Cela rend le gateway accessible sur http://<IP_MASTER>:8080

# Se connecter
cat ~/openfaas-password.txt | faas-cli login --username admin --password-stdin --gateway
$GATEWAY_URL

# Vérifier la connexion
faas-cli version --gateway $GATEWAY_URL
```

📄 Copier

### Définir le gateway par défaut

```
# Ajouter dans ~/.bashrc ou ~/.zshrc sur le master
# Remplacez <IP_MASTER> par l'adresse IP de votre nœud master
export OPENFAAS_URL=http://<IP_MASTER>:$GATEWAY_PORT

# Ou si vous utilisez port-forward :
export OPENFAAS_URL=http://<IP_MASTER>:8080
```

📄 Copier

📄 **Accès depuis le réseau local** : Avec un cluster on-premise, le gateway est accessible via le **NodePort** sur n'importe quel nœud (master ou workers). Pas besoin de port-forward si vous êtes sur le même réseau.

---

## 4. Exploration de l'interface utilisateur OpenFaaS

### Accéder à l'UI

Ouvrez votre navigateur à l'adresse du gateway : `http://<IP_MASTER>:<NODEPORT>/ui/` (ou `http://<IP_MASTER>:8080/ui/` si vous utilisez port-forward).

Connectez-vous avec :

- **Utilisateur** : admin
- **Mot de passe** : celui généré lors de l'installation

### Fonctionnalités de l'UI

L'interface web d'OpenFaaS permet de :

- 📄 **Lister** toutes les fonctions déployées

- **Déployer** de nouvelles fonctions depuis le Function Store
- **► Invoquer** les fonctions directement (avec un body en entrée)
- **📄 Voir les métriques** : nombre d'invocations, réplicas actifs
- **🗑️ Supprimer** des fonctions

## Déployer une fonction depuis le Store (test rapide)

Depuis l'UI :

1. Cliquez sur **"Deploy New Function"**
2. Parcourez le **Function Store**
3. Sélectionnez par exemple `figlet` (convertit du texte en art ASCII)
4. Cliquez sur **"Deploy"**
5. Une fois déployée, sélectionnez la fonction et entrez du texte dans le champ "Request Body"
6. Cliquez sur **"Invoke"**

Ou via le CLI :

```
# Déployer figlet depuis le store
faas-cli store deploy figlet

# Invoquer la fonction
echo "OpenFaaS" | faas-cli invoke figlet

# Résultat :
#
#  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
# / _ \ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
# | | | | ' _ \ / _ \ ' _ \ | / _ \ ' _ \ \ _ \ \
# | | | | | ) | _/ | | | | ( | | ( | | ) |
# \ _/ | . _/ \ _/ | | | \ _/ | \ _/ |
#
#   | |
```

📄 Copier

## 5. Création de fonctions

### Structure d'une fonction OpenFaaS

Chaque fonction OpenFaaS est composée de :

```
fonction1/
├─ handler.py      # Le code de la fonction (langage au choix)
├─ requirements.txt # Dépendances (spécifique au langage)
└─ ...
```

📄 Copier

Et un fichier de configuration à la racine du projet :

```
# fonction1.yaml
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080

functions:
  fonction1:
    lang: python3-http
    handler: ./fonction1
    image: ghcr.io/nouvy/fonction1:latest
```

📄 Copier

**i gateway** : L'adresse du gateway OpenFaaS. Ici `127.0.0.1:8080` car on utilise un port-forward, mais vous pouvez aussi utiliser `http://<IP_MASTER>:<NODEPORT>` directement.

**i image** : L'image doit pointer vers un registry **public** (obligatoire en Community Edition). Ici on utilise **ghcr.io** (GitHub Container Registry).

## Le Template Store

OpenFaaS fournit des templates officiels pour de nombreux langages :

```
# Lister les templates disponibles
faas-cli template store list

# Résultat partiel :
# NAME           SOURCE           DESCRIPTION
# csharp         openfaas        C# template
# dockerfile     openfaas        Dockerfile template
# go             openfaas        Go template
# java11         openfaas        Java 11 template
# node18         openfaas        Node.js 18 template
# php8           openfaas        PHP 8 template
# python3        openfaas        Python 3 template
# python3-http   openfaas        Python 3 with HTTP
# ruby           openfaas        Ruby template

# Télécharger un template
faas-cli template store pull python3-http
```

📄 Copier

## Types de templates

Template	Description	Utilisation
python3	Template classique (stdin/stdout)	Fonctions simples, traitement de texte
python3-http	Template HTTP avec Flask	APIs REST, webhooks, réponses JSON
node18	Node.js 18	APIs JavaScript, intégrations
go	Go natif	Haute performance, microservices
dockerfile	Dockerfile personnalisé	Contrôle total sur l'image

## 6. Développement d'une fonction serverless

### Créer la fonction

```
# Créer un nouveau projet de fonction
faas-cli new fonction1 --lang python3-http

# Structure créée :
# fonction1/
# └─ handler.py
# └─ requirements.txt
# fonction1.yaml
# template/          (templates téléchargés)
```

📄 Copier

### Le code par défaut : handler.py

Le template `python3-http` génère un fichier `fonction1/handler.py` avec le code suivant :

```
def handle(event, context):
    return {
        "statusCode": 200,
        "body": "Hello from OpenFaaS!"
    }
```

📄 Copier

C'est tout ! La fonction reçoit un `event` (la requête HTTP) et retourne une réponse avec un **status code** et un **body**.

### Comprendre event et context

Paramètre	Description
<code>event.body</code>	Le corps de la requête (texte ou bytes)
<code>event.headers</code>	Les headers HTTP de la requête
<code>event.method</code>	La méthode HTTP (GET, POST, etc.)
<code>event.query</code>	Les paramètres de query string
<code>event.path</code>	Le chemin de la requête
<code>context</code>	Contexte d'exécution (rarement utilisé)

### Exemple : personnaliser la fonction

Modifiez `fonction1/handler.py` pour traiter les données en entrée :

```
import json

def handle(event, context):
    """Fonction qui salue l'utilisateur par son nom."""

    body = event.body
    if isinstance(body, bytes):
        body = body.decode("utf-8")

    try:
        data = json.loads(body)
        name = data.get("name", "le monde")
    except (json.JSONDecodeError, ValueError):
        name = body if body else "le monde"

    return {
        "statusCode": 200,
        "body": json.dumps({
            "message": f"Bonjour {name} depuis OpenFaaS !",
            "method": event.method,
            "path": event.path
        }, ensure_ascii=False)
    }
```

📄 Copier

## Ajouter des dépendances

Si votre fonction a besoin de bibliothèques Python, ajoutez-les dans `fonction1/requirements.txt` :

```
requests
```

📄 Copier

Elles seront installées automatiquement lors du build de l'image Docker.

---

## 7. Empaquetage et déploiement de fonctions

### Le fichier `fonction1.yaml`

```
version: 1.0
provider:
  name: openfaas
  gateway: http://<IP_MASTER>:8080

functions:
  fonction1:
    lang: python3-http
    handler: ./fonction1
    image: ghcr.io/nouvy/fonction1:latest
```

📄 Copier

🔗 **Community Edition** : L'image doit être **publique** sur le registry. Pensez à passer le package en public sur GitHub (Package settings > Change visibility > Public).

## Étape 1 : Construire l'image Docker

Le build se fait sur une machine où **Docker est installé** (serveur Docker de build, pas forcément le master Kubernetes).

```
# Construire l'image
faas-cli build -f fonction1.yaml
```

📄 Copier

**i** *Que se passe-t-il ? OpenFaaS génère un Dockerfile basé sur le template python3-http, y ajoute votre code handler.py et les dépendances de requirements.txt, puis construit l'image Docker.*

## Étape 2 : Pousser l'image vers ghcr.io

```
# Se connecter à ghcr.io (une seule fois)
echo "VOTRE_GITHUB_TOKEN" | docker login ghcr.io -u novvy --password-stdin

# Pousser l'image
faas-cli push -f fonction1.yaml
```

📄 Copier

*Le token GitHub doit avoir le scope write:packages. Créez-le dans **GitHub > Settings > Developer settings > Personal access tokens.***

## Étape 3 : Déployer sur OpenFaaS

Depuis le **master Kubernetes** (ou toute machine avec faas-cli et accès au gateway) :

```
# Se connecter au gateway OpenFaaS
export OPENFAAS_URL=http://<IP_MASTER>:<NODEPORT>
cat ~/openfaas-password.txt | faas-cli login --username admin --password-stdin

# Déployer la fonction
faas-cli deploy -f fonction1.yaml
```

📄 Copier

---

## 8. Test de la fonction

### Méthode 1 : Via le CLI

```
# Invoquer la fonction (retourne "Hello from OpenFaaS!")
echo "" | faas-cli invoke fonction1

# Invoquer avec un nom
echo '{"name": "Fabrice"}' | faas-cli invoke fonction1

# Résultat :
# {"message": "Bonjour Fabrice depuis OpenFaaS !", "method": "POST", "path": "/"}
```

📄 Copier

## Méthode 2 : Via curl

```
# Invoquer directement via l'API Gateway
curl -X POST http://<IP_MASTER>:<NODEPORT>/function/fonction1 \
  -H "Content-Type: application/json" \
  -d '{"name": "Fabrice"}'

# Avec authentification
curl -X POST http://<IP_MASTER>:<NODEPORT>/function/fonction1 \
  -u admin:$(cat ~/openfaas-password.txt) \
  -d '{"name": "Fabrice"}'
```

📄 Copier

## Méthode 3 : Via l'UI OpenFaaS

1. Ouvrez `http://<IP_MASTER>:<NODEPORT>/ui/`
2. Sélectionnez la fonction `fonction1`
3. Entrez `{"name": "Fabrice"}` dans **"Request Body"**
4. Cliquez sur **"Invoke"**
5. Consultez la réponse dans **"Response Body"**

## Vérifier le statut des fonctions

```
# Lister toutes les fonctions déployées
faas-cli list

# Résultat :
# Function          Invocations  Replicas
# fonction1         3            1

# Détails d'une fonction
faas-cli describe fonction1
```

📄 Copier