Cloudflare Pages & Workers : Déploiement Serverless Gratuit — Guide 2026

Cloudflare Pages et Workers permettent de déployer des sites statiques et des APIs serverless à la périphérie du réseau mondial de Cloudflare. Ce guide couvre le déploiement CI/CD via Git, les Edge Functions, KV Storage, D1 SQLite et la sécurisation des pipelines dans une approche DevSecOps.

Cloudflare Pages et Workers révolutionnent le déploiement serverless en plaçant le code au plus près des utilisateurs, sur les 300+ PoP (Points of Presence) du réseau Cloudflare répartis dans 100 pays. Cloudflare Workers est un runtime JavaScript/TypeScript et WebAssembly qui exécute le code en moins de 5ms de latence grâce à l'isolation V8 Isolates (plus légère que les conteneurs Docker), sans serveur à gérer, sans cold start (contrairement à AWS Lambda), et avec une tarification à zéro pour les 100 000 premières requêtes par jour. Cloudflare Pages étend cette infrastructure aux sites statiques et aux applications JAMstack (Next.js, Gatsby, Astro, SvelteKit) avec un pipeline CI/CD Git intégré qui déploie automatiquement à chaque push. En 2026, plus de 6 millions de sites utilisent Cloudflare Pages selon les données Cloudflare. Dans une approche DevSecOps, Cloudflare propose des fonctionnalités de sécurité intégrées au pipeline : Cloudflare Access (Zero Trust), WAF intégré, scan de secrets dans les variables d'environnement, et Cloudflare Radar pour la détection des menaces. Ce guide pratique couvre le déploiement d'une application complète avec pipeline CI/CD sécurisé, les fonctions Edge, KV Storage, D1 SQLite et les bonnes pratiques de sécurisation des pipelines dans une démarche DevSecOps.

À RETENIR

À retenir

V8 Isolates vs conteneurs : Cloudflare Workers utilise des isolates V8 (même technologie que Chrome pour sandboxer les onglets), 35x plus légers que les conteneurs Docker — cold start < 1ms, facturation à la requête, pas de serveur à maintenir.

Cloudflare Pages = CI/CD gratuit intégré : connexion GitHub/GitLab, détection automatique du framework (Next.js, Nuxt, Astro, Hugo, etc.), déploiements preview sur chaque Pull Request avec URL unique, et déploiement production automatique sur merge dans main.

KV Storage pour les données edge : Cloudflare KV est un store clé-valeur distribué globalement avec cohérence éventuelle (60s), idéal pour les sessions utilisateurs, les rates limits et les données de configuration — pas adapté aux données transactionnelles.

D1 SQLite pour les données relationnelles : D1 est la base de données SQL de Cloudflare Workers (SQLite à la périphérie), supportant les transactions ACID, les jointures et les index — gratuit jusqu'à 5 millions de lignes en 2026.

Pipeline DevSecOps intégré : les secrets d'environnement Workers sont chiffrés au repos avec AES-256, les Workers peuvent être protégés par Cloudflare Access (Zero Trust SSO), et les Workers Secrets empêchent l'exposition des variables sensibles dans les logs.

Architecture Cloudflare Pages et Workers : comprendre les composants

Cloudflare propose un écosystème cohérent de services edge : Pages pour les sites statiques et les applications frontend, Workers pour la logique backend sans serveur, KV pour le stockage clé-valeur distribué, D1 pour la base de données SQLite relationnelle, R2 pour le stockage d'objets compatible S3 (sans coût de sortie), Queues pour les messages asynchrones, et Durable Objects pour les états partagés entre Workers. Ces composants s'assemblent pour construire des architectures full-stack entièrement serverless, sans gérer un seul serveur.

L'avantage architectural majeur est la latence : un Worker s'exécute dans le PoP Cloudflare le plus proche de l'utilisateur, typiquement à <15ms de latence réseau depuis les grandes métropoles mondiales. AWS Lambda, par comparaison, s'exécute dans une région spécifique (eu-west-1, us-east-1) et ajoute 50 à 200ms de latence pour les utilisateurs éloignés de cette région. Cette approche edge-first est particulièrement pertinente pour les APIs à faible latence, les applications temps réel et les middlewares d'authentification.

Déployer un site statique avec Cloudflare Pages et Git CI/CD

Le déploiement d'un site avec Cloudflare Pages se fait en quelques minutes via l'interface Cloudflare Dashboard ou la CLI Wrangler. La connexion GitHub ou GitLab est requise pour le CI/CD automatique.

```
 # Installation de Wrangler CLI (outil officiel Cloudflare)
npm install -g wrangler

# Authentification Cloudflare
wrangler login
# Ouvre le navigateur pour autoriser l'accès à votre compte Cloudflare

# Créer un nouveau projet Cloudflare Pages
wrangler pages project create mon-site     --production-branch main

# Déployer manuellement (sans CI/CD Git)
# Compiler le projet d'abord (exemple Next.js)
npm run build && npm run export

# Déployer le répertoire out/ (ou dist/ selon le framework)
wrangler pages deploy out/ --project-name mon-site

# URL de déploiement retournée :
# https://abc123.mon-site.pages.dev (preview)
# https://mon-site.pages.dev (production après configuration custom domain)
```

Configuration du pipeline CI/CD sécurisé (approche DevSecOps)

Dans une approche DevSecOps, le pipeline de déploiement Cloudflare Pages doit intégrer des contrôles de sécurité automatiques : scan des secrets dans le code, audit des dépendances, tests de sécurité et protection des environnements de production.

```yaml
 # .github/workflows/deploy-cloudflare-pages.yml
# Pipeline CI/CD sécurisé pour Cloudflare Pages

name: Deploy to Cloudflare Pages (DevSecOps)

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  security-scan:
    name: Security Scanning
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      # Scan des secrets dans le code source (gitleaks)
      - name: Scan for secrets
        uses: gitleaks/gitleaks-action@v2
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

      # Audit des dépendances npm
      - name: NPM Security Audit
        run: |
          npm ci
          npm audit --audit-level=high
          npx better-npm-audit audit

      # SAST basique avec Semgrep
      - name: Semgrep Security Scan
        uses: semgrep/semgrep-action@v1
        with:
          config: >-
            p/javascript
            p/typescript
            p/owasp-top-ten

  build-and-deploy:
    name: Build and Deploy
    needs: security-scan
```

```
    runs-on: ubuntu-latest
    permissions:
      contents: read
      deployments: write

    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js 22
        uses: actions/setup-node@v4
        with:
          node-version: '22'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci --ignore-scripts  # --ignore-scripts évite les supply chain attacks

      - name: Build
        run: npm run build
        env:
          NODE_ENV: production
          # Les secrets sont injectés depuis GitHub Secrets, jamais en dur
          API_KEY: ${{ secrets.API_KEY }}

      - name: Deploy to Cloudflare Pages
        uses: cloudflare/pages-action@v1
        with:
          apiToken: ${{ secrets.CLOUDFLARE_API_TOKEN }}
          accountId: ${{ secrets.CLOUDFLARE_ACCOUNT_ID }}
          projectName: mon-site
          directory: dist/
          gitHubToken: ${{ secrets.GITHUB_TOKEN }}
```

Comment développer des Workers avec KV Storage et D1 ?

Cloudflare Workers s'écrit en JavaScript/TypeScript, en Rust (compilé en WebAssembly), ou en Python (beta). L'API Workers est proche des Web Standards (Fetch API, WebCrypto,

ReadableStream) mais avec des APIs spécifiques Cloudflare pour accéder aux services edge (KV, D1, R2, Queues).

```
 # Initialiser un nouveau Worker avec TypeScript
wrangler init mon-worker --type=typescript

# Structure du projet généré :
# mon-worker/
# ├── src/
# │   └── index.ts       (code principal)
# ├── wrangler.toml      (configuration Wrangler)
# ├── package.json
# └── tsconfig.json
```

```
 # wrangler.toml — Configuration du Worker avec KV et D1
cat > wrangler.toml << 'EOF'
name = "mon-api-worker"
main = "src/index.ts"
compatibility_date = "2026-07-01"
compatibility_flags = ["nodejs_compat"]

# Binding KV Storage
[[kv_namespaces]]
binding = "SESSION_STORE"
id = "VOTRE_KV_NAMESPACE_ID"

# Binding D1 Database (SQLite)
[[d1_databases]]
binding = "DB"
database_name = "prod-database"
database_id = "VOTRE_D1_DATABASE_ID"

# Binding R2 Storage
[[r2_buckets]]
binding = "ASSETS"
bucket_name = "mon-bucket-assets"

# Variables d'environnement (non-secrets)
[vars]
ENVIRONMENT = "production"
API_VERSION = "v2"

# Secrets (définis via `wrangler secret put NOM_SECRET`)
# Ne jamais mettre les secrets dans wrangler.toml !
EOF
```

```
 # Exemple de Worker TypeScript avec D1 et KV
cat > src/index.ts << 'EOF'
interface Env {
  DB: D1Database;
  SESSION_STORE: KVNamespace;
  API_SECRET: string;
}

export default {
  async fetch(request: Request, env: Env, ctx: ExecutionContext): Promise {
    const url = new URL(request.url);

    // Route : GET /api/users
    if (url.pathname === '/api/users' && request.method === 'GET') {
      // Vérification du token (secret injecté via wrangler secret)
      const authHeader = request.headers.get('Authorization');
      if (!authHeader || authHeader !== `Bearer ${env.API_SECRET}`) {
        return new Response('Unauthorized', { status: 401 });
      }

      // Vérifier le cache KV (TTL: 60 secondes)
      const cacheKey = 'users_list';
      const cached = await env.SESSION_STORE.get(cacheKey);
      if (cached) {
        return new Response(cached, {
          headers: { 'Content-Type': 'application/json', 'X-Cache': 'HIT' }
        });
      }

      // Requête D1 (SQLite edge)
      const { results } = await env.DB.prepare(
        'SELECT id, name, email, created_at FROM users ORDER BY created_at DESC LIMIT 50'
      ).all();

      const responseData = JSON.stringify(results);

      // Stocker en cache KV pour 60 secondes
      await env.SESSION_STORE.put(cacheKey, responseData, { expirationTtl: 60 });

      return new Response(responseData, {
        headers: { 'Content-Type': 'application/json', 'X-Cache': 'MISS' }
      });
    }
```

```
    return new Response('Not Found', { status: 404 });
  }
};
EOF
```

Gestion des secrets et sécurisation des Workers

La gestion des secrets dans Cloudflare Workers suit les principes DevSecOps : les secrets ne doivent jamais figurer dans le code source, les fichiers wrangler.toml ou les variables d'environnement non chiffrées. Cloudflare Workers Secrets chiffre les valeurs au repos avec AES-256 et les injecte dans l'environnement d'exécution du Worker sans les exposer dans les logs.

```
 # Définir des secrets (chiffrés dans Cloudflare, jamais visibles en clair)
wrangler secret put DATABASE_PASSWORD
# Entrer la valeur quand demandé (non affiché à l'écran)

wrangler secret put API_SECRET
wrangler secret put STRIPE_SECRET_KEY

# Lister les secrets définis (les valeurs ne sont pas visibles)
wrangler secret list
# Output :
# Name               Created
# DATABASE_PASSWORD  2026-07-01T10:00:00Z
# API_SECRET         2026-07-01T10:00:01Z
# STRIPE_SECRET_KEY  2026-07-01T10:00:02Z

# Supprimer un secret compromis
wrangler secret delete API_SECRET_OLD

# Rotation de secret (remplacer la valeur)
wrangler secret put API_SECRET  # Entrer la nouvelle valeur
```

> En migration d'une API Node.js hébergée sur un VPS Hetzner vers Cloudflare Workers pour une startup SaaS (50 000 requêtes/jour), les performances mesurées montrent un gain de latence de 180ms à 8ms en P50 pour les utilisateurs européens, et de 320ms à 15ms pour les utilisateurs en Asie-Pacifique. Le coût mensuel est passé de 24€ (VPS) à 0€ (plan gratuit Workers). Le principal défi : réécrire les middlewares Express en APIs Web Standard (pas de `req.headers` mais `request.headers.get()`). La migration a pris 2 jours. Point de vigilance DevSecOps : les Workers Logs (Cloudflare) journalisent les requêtes — s'assurer de ne jamais logguer de données sensibles (tokens, mots de passe) dans le code Worker.
>
> *— Retour de migration serverless Cloudflare Workers, avril 2026*

Cloudflare Pages Functions : le backend intégré aux pages statiques

Cloudflare Pages Functions permet d'ajouter une logique backend directement dans un projet Pages, sans créer un Worker séparé. Les fonctions se placent dans le répertoire `functions/` et suivent un routage basé sur le système de fichiers, similaire à Next.js App Router ou Remix.

```
 # Structure d'un projet Pages avec Functions
# mon-site/
# ├── dist/           (build statique)
# ├── functions/
# │   ├── api/
# │   │   ├── [[path]].ts    (catch-all pour /api/*)
# │   │   └── users.ts       (route /api/users)
# │   └── _middleware.ts     (middleware global)
# └── wrangler.toml

cat > functions/api/users.ts << 'EOF'
import { PagesFunction } from "@cloudflare/workers-types";

interface Env {
  DB: D1Database;
}

export const onRequestGet: PagesFunction = async (context) => {
  const { results } = await context.env.DB.prepare(
    "SELECT id, name FROM users LIMIT 20"
  ).all();

  return Response.json(results);
};

export const onRequestPost: PagesFunction = async (context) => {
  const body = await context.request.json<{name: string, email: string}>();

  if (!body.name || !body.email) {
    return new Response("Missing required fields", { status: 400 });
  }

  await context.env.DB.prepare(
    "INSERT INTO users (name, email) VALUES (?, ?)"
  ).bind(body.name, body.email).run();

  return Response.json({ status: "created" }, { status: 201 });
};
EOF

# Tester localement avec wrangler pages dev
wrangler pages dev dist/ --binding DB=db --local
```

Comparaison : Cloudflare Workers vs AWS Lambda vs Vercel Edge

Critère	Cloudflare Workers	AWS Lambda@Edge	Vercel Edge Functions
Runtime	V8 Isolates (JS/TS/Wasm)	Node.js/Python/Go/Java	V8 Isolates (JS/TS)
Cold start	<1ms	100-500ms	<1ms
Latence réseau	300+ PoP mondiaux	13 régions CloudFront	Vercel CDN (~100 PoP)
Plan gratuit	100K req/jour	Gratuit inclus CF	100K req/jour
D1/SQLite edge	Oui (natif)	Non (DynamoDB distant)	Oui (via Vercel KV)
Taille max bundle	10 MB (Workers) / 1 MB (Pages)	50 MB (zippé)	4 MB
Accès npm complet	Partiel (Edge runtime)	Oui (Node.js complet)	Partiel (Edge runtime)

Bonnes pratiques DevSecOps pour les Workers Cloudflare

La sécurisation d'un déploiement Cloudflare Workers en production nécessite plusieurs couches de protection complémentaires aux contrôles CI/CD déjà évoqués.

```
 # 1. Restreindre l'accès à l'API Admin Cloudflare
# Créer des API Tokens avec permissions minimales (principe du moindre privilège)
# Cloudflare Dashboard > Profile > API Tokens > Create Token
# Permissions minimales pour le déploiement uniquement :
# - Account > Cloudflare Pages > Edit
# - Zone > Workers Routes > Edit (si routes custom)

# 2. Activer Cloudflare Access pour protéger les environments preview
# Zero Trust > Access > Applications > Add an Application
# Protéger *.mon-site.pages.dev avec SSO (Google, Okta, Microsoft Entra ID)

# 3. Rate Limiting au niveau Workers (protection DDoS applicatif)
# Dans le code Worker :
const rateLimiter = {
  async check(env, ip) {
    const key = `rl:${ip}`;
    const count = parseInt(await env.SESSION_STORE.get(key) || '0');
    if (count > 100) return false;
    await env.SESSION_STORE.put(key, String(count + 1), { expirationTtl: 60 });
    return true;
  }
};

# 4. Content Security Policy (CSP) via Headers Transform
# Dans _headers (Cloudflare Pages) ou via Workers :
cat > dist/_headers << 'EOF'
/*
  X-Frame-Options: DENY
  X-Content-Type-Options: nosniff
  Referrer-Policy: strict-origin-when-cross-origin
  Permissions-Policy: camera=(), microphone=(), geolocation=()
  Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline'; img-src 'self'
EOF
```

La documentation officielle Cloudflare Workers est disponible sur developers.cloudflare.com/workers/. Pour les aspects DevSecOps des pipelines CI/CD, l'article sur les attaques sur les pipelines CI/CD GitHub détaille les vecteurs d'attaque supply chain qui s'appliquent également aux pipelines Cloudflare Pages. La gestion des secrets Workers s'inscrit dans une politique plus large de lutte contre le secrets sprawl en environnement cloud. Pour les environnements Kubernetes qui coexistent avec les Workers edge, l'article sur la sécurité Kubernetes présente des outils complémentaires.

Questions fréquentes

Cloudflare Workers supporte-t-il Node.js ?

Cloudflare Workers supporte partiellement Node.js via le flag de compatibilité `nodejs_compat` dans wrangler.toml. Ce mode implémente les modules Node.js les plus courants (path, crypto, buffer, stream, util, events) mais pas tous — en particulier, les modules natifs comme `fs` (système de fichiers) et `child_process` ne sont pas disponibles dans le runtime edge. Pour les applications Node.js existantes, il est généralement nécessaire de réécrire les parties incompatibles ou d'utiliser des alternatives edge-first. L'alternative est Cloudflare Workers for Platforms qui offre un runtime Node.js plus complet.

Comment gérer les variables d'environnement en local avec Wrangler ?

En développement local, Wrangler utilise le fichier `.dev.vars` pour les secrets (équivalent local des Workers Secrets). Ce fichier doit être dans `.gitignore` et ne jamais être commité. Les variables non-sensibles peuvent être définies dans `wrangler.toml` sous `[vars]`. En CI/CD, les secrets sont injectés depuis les secrets du dépôt (GitHub Secrets, GitLab CI Variables) via l'action `cloudflare/pages-action`. La commande `wrangler pages dev --binding VAR=valeur` permet aussi d'injecter des variables au démarrage du serveur de développement local.

Quelle est la limite de taille d'un Worker Cloudflare ?

Un Worker compilé (après minification et bundling) est limité à 10 MB sur le plan payable (Workers Paid, 5$/mois). Sur le plan gratuit, la limite est de 1 MB. Pour Cloudflare Pages Functions, la limite est de 1 MB par fichier. Ces limites concernent le bundle JavaScript/Wasm compilé — la plupart des Workers TypeScript compilés restent sous 500 KB après minification. Pour les Workers volumineux (ex: avec une librairie ML WebAssembly), utiliser Workers Unbound (plan entreprise) ou optimiser le bundle avec des imports dynamiques.

Comment migrer une application Express.js vers Cloudflare Workers ?

La migration d'Express vers Workers nécessite de remplacer l'API Express par l'API Fetch API standard. Le framework `Hono` (disponible sur npm) est le successeur naturel d'Express pour les environments edge : sa syntaxe est proche d'Express (`app.get('/', (c) => c.text('Hello'))`), il s'exécute nativement sur Workers, Bun, Deno et Node.js. Hono gère le routage, les middlewares, les validations et le rendu JSX, couvrant les cas d'usage Express les plus courants. La migration d'Express 5 vers Hono prend généralement 1 à 3 jours pour une API de taille moyenne.

Cloudflare D1 est-il adapté pour une application en production en 2026 ?

Oui, D1 est en GA (General Availability) depuis mars 2024 et est adapté à la production en 2026. D1 supporte les transactions ACID, les index, les jointures et les mises à jour en masse. Les limites actuelles : 500 Mo par base de données (plan gratuit), 50 Go (plan payant Workers), lecture et écriture depuis un seul point de présence Cloudflare (réplication en lecture vers 10+ régions). D1 n'est pas adapté aux charges d'écriture très importantes (>1000 writes/seconde) — pour ces cas, Cloudflare Durable Objects ou une base de données relationnelle managée (PlanetScale, Neon.tech) sont préférables.

Observabilité et logs dans Cloudflare Workers

L'observabilité est un enjeu spécifique dans les architectures serverless edge. Cloudflare Workers Logs (Tail Workers) permet de capturer et analyser les logs de tous les Workers en temps réel. Chaque requête Workers peut être journalisée vers un Tail Worker qui traite et expédie les logs vers un backend d'observabilité (Grafana Cloud, Datadog, New Relic, ou un bucket R2 pour l'archivage longue durée).

```
 # Tail Worker : collecte des logs en temps réel
# wrangler tail mon-api-worker
# Affiche en temps réel toutes les requêtes et leurs logs

# Dans le code Worker : utilisation de console.log pour les logs (captés par Tail)
# Exemple dans TypeScript :
# console.log(JSON.stringify({
#   timestamp: new Date().toISOString(),
#   method: request.method,
#   path: new URL(request.url).pathname,
#   duration_ms: Date.now() - startTime,
#   status: response.status,
#   cf_ray: request.headers.get('cf-ray')
# }))

# Workers Analytics Engine : métriques personnalisées
# env.ANALYTICS.writeDataPoint({
#   indexes: ['mon-worker'],
#   doubles: [duration, responseSize],
#   blobs: [country, path]
# })
```

Cloudflare Workers et Zero Trust (Access)

Cloudflare Access (composant Zero Trust) permet de protéger les Workers et Pages deployments avec une couche d'authentification SSO sans modifier le code de l'application. Pour les environnements preview (chaque Pull Request génère un URL unique), Cloudflare Access garantit que seuls les membres de l'équipe peuvent accéder aux déploiements de préproduction.

```
 # Configurer Cloudflare Access pour protéger les URLs preview
# Via l'interface Cloudflare Zero Trust :
# Zero Trust > Access > Applications > Add Application
# Application type : Self-hosted
# Application URL : *.mon-site.pages.dev
# Policy : Allow
#   Rule type : Emails ending in @domaine.com

# Via la CLI Cloudflare (Terraform) :
# resource 'cloudflare_access_application' 'preview' {
#   zone_id          = var.cloudflare_zone_id
#   name             = 'Preview Deployments'
#   domain           = '*.mon-site.pages.dev'
#   session_duration = '24h'
#   logo_url         = 'https://mon-site.com/logo.png'
# }
```

Pour les aspects DevSecOps des pipelines CI/CD dans lesquels s'intègre Cloudflare Pages, l'article sur les attaques sur les pipelines CI/CD GitHub détaille les vecteurs supply chain qui s'appliquent à tous les pipelines, y compris Cloudflare. La gestion des secrets Workers s'inscrit dans une politique globale de lutte contre le secrets sprawl en environnement cloud. Cloudflare Workers s'intègre également dans les architectures de protection des applications cloud-native (CNAPP) comme WAF et API gateway distribué.

Cloudflare Pages et Workers s'intègrent dans une architecture DevSecOps globale où la sécurité est intégrée dès la conception. Les Workers Secrets, les pipelines CI/CD avec scan de secrets, les politiques Cloudflare Access Zero Trust et les règles WAF forment ensemble une stratégie de défense en profondeur pour les applications edge. Pour une vue complète de la sécurisation des pipelines CI/CD dans lesquels s'inscrit Cloudflare Pages, l'article sur les attaques sur les pipelines GitHub est complémentaire. La documentation complète de l'API Cloudflare Workers est disponible sur developers.cloudflare.com/workers/runtime-apis/.

Les fonctionnalités avancées de Cloudflare Workers (Durable Objects pour l'état partagé, Queues pour les files de messages, Pages Functions pour le backend intégré) permettent de construire des applications full-stack serverless à l'échelle mondiale, sans compromis sur les performances ni la sécurité des secrets. Cloudflare continue d'innover dans l'espace edge computing, avec des fonctionnalités comme Workers AI (inférence LLM à la périphérie) et Browser Rendering API

(Chromium headless dans un Worker) qui élargissent encore le périmètre des cas d'usage possibles dans les architectures DevSecOps modernes.

