Recherche plein texte sur un site statique : Pagefind sous IIS, sans CDN et sous CSP stricte

Retour d'expérience — les points durs rencontrés en conditions réelles, et leurs solutions. Cas d'usage : le miroir statique multilingue d'un site vitrine industriel (~250 pages), hébergé sur IIS, avec une politique stricte de confidentialité : aucun service externe, aucun cookie, aucun terme de recherche transmis à un tiers.

1. Pourquoi Pagefind, et l'architecture retenue

Pagefind (CloudCannon) indexe un site statique au moment de sa construction et exécute la recherche entièrement dans le navigateur : un moteur WebAssembly charge à la demande de petits fragments d'index. Ni base de données, ni backend, ni requête sortante — idéal quand la confidentialité est une exigence et non une option. L'installation la plus simple passe par le wrapper Python officiel :

```
pip install "pagefind[extended]" # extended = segmentation CJK incluse
python -m pagefind --site chemin/du/site
```

Choix structurants qui ont bien vieilli sur ce projet :

- **Un seul passage d'indexation** à la racine : Pagefind sépare lui-même les index par langue d'après l'attribut lang de <html>. Ajouter une langue ne demande rien côté moteur.
- **Un glob restreint aux dossiers de langue** ({fr,en,nl,de}/**/*.html) : les répertoires techniques présents à la racine ne polluent jamais l'index.
- **Le périmètre par attributs plutôt que par exclusions** : dès qu'une page porte data-pagefind-body, toutes celles qui ne le portent pas sortent de l'index. Poser l'attribut sur le conteneur de contenu principal exclut d'un coup en-têtes, menus, pieds de page… et toutes les pages utilitaires (remerciements, pages internes) — sans aucune liste noire à maintenir.
- **Des scripts idempotents** : l'injection des attributs est réexécutable et réécrit ses propres métadonnées à chaque passage — indispensable dans un pipeline qui régénère le site.

Deux raffinements de pertinence qui changent tout sur un site produit : ignorer les blocs « produits similaires » (sinon chaque référence croisée matche sur toutes les pages qui la citent) via data-pagefind-ignore, et porter le code produit en métadonnée (data-pagefind-meta="product-code:XYZ-123") — affiché sous chaque résultat.

2. Point dur n°1 — IIS et les extensions inconnues : le 404 déguisé

Symptôme : la recherche « mouline » indéfiniment, puis « Pagefind: WASM Error (No pointer) ».

Le bundle Pagefind contient des fichiers .pf_meta, .pf_index, .pf_fragment, .pf_filter et .wasm. IIS refuse de servir toute extension sans type MIME déclaré — et si le site a une page d'erreur personnalisée (httpErrors avec existingResponse="Replace"), le 404 est remplacé par… du HTML, que le moteur tente de décoder comme un index binaire. D'où le « No pointer », particulièrement trompeur. Piège dans le piège : **.pf_filter n'apparaît que lorsque des filtres existent** — tout fonctionne, puis on ajoute des filtres, et une 404 surgit sur une extension jamais vue.

Solution : un web.config posé dans le dossier du bundle, avec le motif remove+mimeMap (qui évite le 500 « duplicate entry » si l'extension est déjà déclarée plus haut) :

```
<staticContent>
  <remove fileExtension=".pf_meta" />
  <mimeMap fileExtension=".pf_meta" mimeType="application/octet-stream" />
  … idem pour .pf_index, .pf_fragment, .pf_filter, .pf_sort, .pagefind …
  <remove fileExtension=".wasm" />
  <mimeMap fileExtension=".wasm" mimeType="application/wasm" />
</staticContent>
```

Et comme l'option --clean de l'indexeur supprime le dossier du bundle — web.config compris —, faites recopier ce fichier par le script de build depuis une source de vérité, à chaque indexation. Astuce de traçabilité : ajoutez-y un en-tête HTTP maison (ex. X-Search-Config: 1.3) — un simple curl -I sur un fichier du bundle dit alors quelle version est réellement déployée.

3. Point dur n°2 — CSP, WebAssembly… et le Web Worker qui n'hérite de rien

Symptôme : « CompileError: call to WebAssembly.instantiate() blocked by CSP ».

Sous une CSP stricte, la compilation WebAssembly exige la directive **'wasm-unsafe-eval'** dans script-src. Son nom fait peur pour rien : elle n'autorise QUE le WASM, pas eval() JavaScript. Premier réflexe sain : la scoper au plus juste, par exemple avec un <location path="fr/search.html"> qui n'enrichit la CSP que pour la page de recherche.

Et c'est là que le piège se referme : **ça ne suffit pas**. Pagefind exécute son moteur dans un Web Worker, et un worker n'hérite pas de la CSP du document — il obéit à la CSP livrée avec la réponse HTTP **de son propre script** (pagefind-worker.js). Le document est autorisé, le worker reste bloqué, et l'erreur persiste alors que curl montre le bon en-tête sur la page. Le diagnostic qui départage en une commande :

```
curl -s -I https://exemple.tld/pagefind/pagefind-worker.js | findstr -i content-security
```

Solution élégante : porter la CSP enrichie par le web.config **du dossier du bundle** (le même fichier que les types MIME). Elle accompagne alors toutes les réponses du dossier — worker compris — et voyage avec le bundle dans tous les environnements. Le reste du site conserve sa CSP stricte à l'identique.

Corollaire vécu : si un cache client agressif est configuré sur les pages HTML (clientCache), le navigateur peut rejouer une page avec son ANCIEN en-tête CSP après votre correctif. Toujours vérifier au curl (qui ne cache pas), puis recharger avec le cache désactivé.

4. Point dur n°3 — lang="fr-FR" n'est pas lang="fr"

Symptôme : « Aucun résultat » sur une page de test… alors que l'index est plein.

Pagefind crée un index par valeur de l'attribut lang, **sans normaliser** : fr-FR devient l'index « fr-fr », distinct d'un éventuel « fr ». Si le gros du site est en fr-FR mais que deux pages utilitaires portent lang="fr" — et qu'une page de test est écrite avec lang="fr" — la recherche interroge le micro-index de deux pages. Le fichier pagefind-entry.json du bundle liste les index réellement créés : faites-le lire par votre script de build et alertez sur toute langue inattendue (des pages de remerciement en de/en dans un dossier fr, ça arrive). La stratégie data-pagefind-body règle d'ailleurs souvent le problème à la racine, en excluant ces pages-là.

5. Point dur n°4 — le mode « bouton » et le debounce qui avale tout

Symptôme : clic sur « Rechercher » → strictement rien. Ni erreur console, ni requête réseau.

Besoin d'ergonomie : une recherche déclenchée au clic ou à Entrée, pas à chaque frappe. Idée naïve : garder l'UI par défaut et pousser debounceTimeoutMs à une valeur énorme, puis appeler triggerSearch() depuis un bouton. Or la lecture du source (minifié) de pagefind-ui révèle :

```
I>0&&Q; ? (clearTimeout(de), de=setTimeout(Y,I), …) : Y()
```

Dès que le debounce est > 0, **toute** recherche passe par le minuteur — triggerSearch() compris. Le clic était donc programmé… pour 24 heures plus tard, en silence parfait. Le vrai mode bouton s'obtient à l'envers :

- **debounceTimeoutMs: 0** — la même ligne montre qu'à zéro, la recherche est immédiate ;
- masquer le champ interne de l'UI (c'est lui qui écoute la frappe) — attention, son pictogramme loupe est un pseudo-élément du formulaire, à masquer aussi ;
- présenter son propre champ, relié à rien : taper ne déclenche rien ;
- bouton et touche Entrée appellent ui.triggerSearch(valeur) — instantané ;
- recréer le bouton « effacer » (le natif suit le champ masqué) en relayant le clear interne de l'UI.

Deux bonus du même chantier : un lien profond ?q=terme (lu au chargement, triggerSearch au démarrage) et la position mesurée sur les **URLs uniques** dans les tests — avec les sous-résultats (ancres) activés, un seul résultat

produit jusqu'à trois liens, ce qui fausse toute mesure naïve de classement.

6. Trois pièges plus petits, mais coûteux

Le poids inline qui fuit dans les titres.
Envelopper un mot du H1 dans pour booster un code produit fait fuir le marqueur interne ___END_PAGEFIND_WEIGHT___ dans les titres de résultats (le titre est dérivé du H1). Ne jamais poser de poids inline dans l'élément source du titre ; le H1 a déjà un poids natif élevé (7), et l'identification exacte passe très bien par une métadonnée.

Deux filtres dans un attribut = un seul filtre composite.
data-pagefind-filter="type:Produit, categorie:X" crée UN filtre dont la valeur est « Produit, categorie:X ». Un attribut par filtre, sur des éléments différents si besoin (le type sur le conteneur, la catégorie sur le H1).

Le déploiement qui exclut trop.
Si votre publication copie le site avec une exclusion globale des web.config (classique : le web.config de production est géré à part), elle exclut AUSSI celui du bundle — et chaque mise en production casse la recherche. Une recopie ciblée du dossier du bundle après coup, et des contrôles post-publication (fichiers présents + en-tête de version testé en HTTP réel) verrouillent définitivement le sujet.

7. Valider sans navigateur : lire l'index lui-même

Les fragments du bundle sont du gzip précédé d'un petit préfixe : une fois décodés, chacun expose l'URL, le contenu texte, les métadonnées et les filtres de sa page — de quoi bâtir un audit statique complet (les requêtes clés trouvent-elles au moins une page ? la fiche attendue est-elle dans l'index avec la bonne métadonnée ? aucune URL de préproduction n'a fui ?) sans lancer un seul navigateur :

```
import gzip, json
d = gzip.decompress(open("fragment/fr-fr_xxx.pf_fragment","rb").read())
if d.startswith(b"pagefind_dcd"): d = d[len(b"pagefind_dcd"):]
page = json.loads(d) # url, content, meta, filters, word_count, anchors
```

Le classement réel, lui, se vérifie en conditions réelles (Playwright) : saisir chaque code, mesurer la position de la fiche exacte, cliquer et exiger un HTTP 200, capturer les erreurs console, tester un viewport mobile, et signaler toute requête sortant du domaine — c'est ce dernier contrôle qui prouve, run après run, la promesse de confidentialité.

8. En résumé — la checklist des points durs

Piège	Parade
IIS : 404 silencieux sur .pf_* (dont .pf_filter, tardif)	web.config MIME dans le dossier du bundle, recopié après chaque --clean, versionné par en-tête HTTP
CSP bloque le WASM… du Worker	'wasm-unsafe-eval' porté par le web.config du bundle (le worker obéit à la CSP de SON script)
fr-FR ≠ fr : index séparés	lang cohérent partout ; lire pagefind-entry.json au build et alerter
debounce > 0 avale triggerSearch()	mode bouton = debounce 0 + champ découplé + triggerSearch
Poids inline dans le H1	jamais ; métadonnée + poids natif du H1
Filtres combinés dans un attribut	un attribut par filtre
Publication qui exclut les web.config	recopie ciblée du bundle + contrôles post-publication en HTTP réel
Cache client : ancienne CSP rejouée	diagnostiquer au curl, recharger cache désactivé

Au bout du chemin : une recherche instantanée, entièrement servie par son propre domaine, qui ne transmet rien à personne, validée par un audit reproductible — et un moteur qui n'a plus de secret pour l'équipe qui l'exploite. C'était le but.

Environnement du retour d'expérience : site statique ~250 pages, IIS, Pagefind 1.5.2 Extended (wrapper Python officiel), UI par défaut (Default UI). Juillet 2026.