WEGLOT TRANSATION CODE GENERATION

Ideo-Lab

22 th of JANUARY 2026

v 1.7

WEGLOT - Pour & Contre

POUR

Pourquoi Weglot est considéré comme un leader

1. Intégration ultra-simple

- Fonctionne par **script JS** ou **plugin** (WordPress, Webflow, Shopify, etc.).
- Aucun refactoring lourd du site existant.
- Compatible SPA, React, Vue, CMS, sites statiques.

2. Détection automatique du contenu

- Analyse le DOM, détecte les chaînes traduisibles.
- Gestion des contenus dynamiques (AJAX, JS runtime).
- Mapping URL multilingue (`/en` , `/fr` , etc.).

3. Qualité de traduction

- S'appuie sur de très bons moteurs (Google / DeepL selon les langues).
- Post-édition humaine possible via une interface dédiée.
- Mémoire de traduction + glossaire.

4. SEO multilingue clé en main

- URLs propres par langue.
- Balises `hreflang` , `meta` , sitemap multilingue.
- Indexation propre par Google (gros point fort).

5. Cible business claire

- PME, e-commerce, scale-ups, agences.
- Temps de mise en œuvre quasi immédiat → ROI rapide.
- Support et doc très solides.

CONTRE

Mais… ses limites (et pourquoi ton projet Ideo-Lab est pertinent)

Soyons honnêtes 😏

- 💸 **Coût élevé** dès qu'on scale (volume de mots / trafic).
- 🔒 **SaaS fermé** : peu de contrôle fin côté moteur, cache, logique.
- ⚙️ **Peu personnalisable** pour des architectures très spécifiques.
- 🧠 Pas pensé pour des **pipelines custom IA / LLM / batch massif**.

👉 C'est exactement là où **ton clone Weglot + moteur i18n Ideo-Lab** devient intéressant :

- contrôle total du parsing,
- choix des moteurs (Google, DeepL, LLM, offline),
- déduplication agressive,
- cache intelligent,
- coûts maîtrisés,
- intégration profonde Django / JS.

En résumé

- **Weglot** = **référence du marché**, très bien conçu, très efficace.
- **Mais pas magique**, ni économique à grande échelle.
- Pour un **projet tech avancé**, industriel, custom → **ta voie est clairement légitime**.

CREATION DU CODE

0) Arborescence proposée

markdown

Copier le code

```
i18n_engine/
  __init__.py
  apps.py
  urls.py
  views.py
  models.py
  services/
    __init__.py
    normalize.py
    dedup.py
    engine.py
    rules.py
```

Structure du Projet

Plaintext

```
ideo_backend/
├── manage.py
├── ideo_backend/          # Configuration globale
│   ├── settings.py
│   └── urls.py
└── translator/            # Notre application principale
    ├── migrations/
    ├── admin.py           # Interface pour visualiser/catégoriser
    ├── apps.py
    ├── models.py          # Le schéma de base de données
    ├── serializers.py     # Validation des données entrantes (du JS)
    ├── urls.py            # Routes API
    ├── views.py           # La logique (Controller)
    └── services/          # Moteurs logiques
        ├── __init__.py
        ├── hasher.py      # Pour créer des IDs uniques par texte
        └── engine.py      # Le futur moteur de traduction
```

Pourquoi Weglot est considéré comme un leader

1. Intégration ultra-simple

- Fonctionne par **script JS** ou **plugin** (WordPress, Webflow, Shopify, etc.).
- Aucun refactoring lourd du site existant.
- Compatible SPA, React, Vue, CMS, sites statiques.

2. Détection automatique du contenu

- Analyse le DOM, détecte les chaînes traduisibles.
- Gestion des contenus dynamiques (AJAX, JS runtime).
- Mapping URL multilingue (`/en` , `/fr` , etc.).

3. Qualité de traduction

- S'appuie sur de très bons moteurs (Google / DeepL selon les langues).
- Post-édition humaine possible via une interface dédiée.
- Mémoire de traduction + glossaire.

4. SEO multilingue clé en main

- URLs propres par langue.
- Balises `hreflang` , `meta` , sitemap multilingue.
- Indexation propre par Google (gros point fort).

5. Cible business claire

- PME, e-commerce, scale-ups, agences.
- Temps de mise en œuvre quasi immédiat → ROI rapide.
- Support et doc très solides.

TO DO

À implémenter (côté JS) :

1. **Marquage** des nœuds traduits
   - ex: `data-i18n-done="1"` sur les éléments, ou un `WeakSet` en mémoire
2. **Fingerprint** du texte source
   - ex: hash simple (crc32/sha1 light) stocké en `data-i18n-hash`
3. **Skip logique** au scan :
   - si `data-i18n-done=1` ET hash identique → **skip**
4. **MutationObserver filtré**
   - ignorer les mutations causées par ta propre traduction (flag "isApplyingTranslations")

Résultat attendu :

- le compteur d'analyse s'arrête (ou devient quasi nul)
- les appels API tombent à ~0 après stabilisation

Ce que je te propose tout de suite

Si tu me donnes **le fichier actuel** `ideolab-i18n.js` (copier-coller ou upload), je te fais un **patch précis** :

- ajout `WeakSet translatedNodes`
- ajout `hashText()` simple
- ajout `markTranslated(el, hash)`
- modification `scanAndTranslate()` pour **skipper**
- verrou `this.isApplying = true/false` autour de l'application
- MutationObserver : **bail out** si `isApplying`

➡️ Et je te renvoie :

- le **fichier complet régénéré** (comme tu préfères)
- - une mini section "comment vérifier" (2 commandes + 2 logs)

AGENDA TO DO

1) Ce que Weglot propose (état des lieux, “blocs de fonctionnalités”)

2) Mapping “Weglot → nous”

3) Ton “Weglot Light” : ce que vous avez déjà vs ce qui manque le plus

1) Ce que Weglot propose (état des lieux, “blocs de fonctionnalités”)

A. Détection & traduction automatique “always-on”

- **Scan / détection de contenu** sur tout le site + traduction initiale (ils parlent de “content detection”, “automatic content detection”). weglot.com +1
- **Traduction continue** : dès qu’un contenu change/une page apparaît, ça repart en détection + traduction. weglot.com +1
- Gestion des **contenus dynamiques** (avec des guides dédiés). support.weglot.c… +1

👉 *Votre “Weglot Light” a déjà une partie de ça (scan + traduction), mais pas encore l’équivalent “toujours synchronisé” proprement (et sans re-calls).*

B. Dashboard de gestion des traductions (TMS simplifié)

- Interface pour **voir/éditer les traductions**, généralement organisée par URL / langue. weglot.com +1
- **Visual Editor** : édition "in context" dans une prévisualisation live du site. weglot.com +1
- **Gestion de tâches / workflow** (le marketing Weglot pousse "auto-pilot", prioriser les pages visibles, etc.). weglot.com +1
- **Collaboration** (multi-utilisateurs/équipe) selon intégrations/plans (souvent mentionné dans leurs contenus comparatifs). brixtemplates.com +1

👉 *Vous avez sûrement : stockage + affichage minimal. Il manque : visual editor + workflow + permissions + outils de tri/filtre.*

C. Glossary & Translation Memory (qualité + cohérence + coût)

- **Glossaire / règles** : protéger des termes ("ne pas traduire X", "toujours traduire Y par Z"). weglot.com +2
- **Translation Memory** : réutiliser les traductions existantes pour éviter retraductions. weglot.com +1

👉 *C'est LE levier direct pour ton problème actuel (des milliers d'appels inutiles) : TM + cache + dédup + invalidation propre.*

D. Exclusions & règles de traduction (contrôle fin)

- **Exclusions** (pages, blocs, sélecteurs, patterns) pour ne pas traduire certaines zones. weglot.com +1
- (En pratique Weglot supporte beaucoup de "rules", par sélecteurs/URLs/éléments — ils l'annoncent surtout via "Exclusions"). weglot.com

👉 *Vous avez déjà commencé à parler de "scopes" côté parser : Weglot formalise ça en règles éditables.*

E. Language switcher (UX)

- **Sélecteur de langue** personnalisable (flags/labels/design) + intégration selon CMS. weglot.com +1

👉 *Côté UI vous êtes en plein dedans : dropdown propre + langue courante sur la même ligne, sans "truc moche".*

F. Multilingual SEO (gros différenciateur)

- **URLs dédiées par langue** (subdomain ou subdirectory) pour indexation. weglot.com +1
- **hreflang auto**, + gestion SEO associée. weglot.com +2
- Weglot insiste que ce n'est pas "juste du JS" : ils mettent en avant l'approche "server-side / embedded in source" quand c'est configuré en subdomain/subdirectory. support.weglot.c... +1
- **Sitemap & SEO meta** (ils expliquent la logique sitemap/hreflang). support.weglot.c... +1

👉 *Si vous voulez être "Weglot-like" commercialement : le SEO multi-langues est un module à part entière, pas juste "on traduit le DOM".*

G. Import / Export

- **Export/import** des traductions pour bosser hors dashboard puis réimporter. weglot.com

👉 *Facile à livrer en V1.5 : CSV/JSON + API endpoints, ça fait très "pro".*

H. Modèle économique (à connaître pour votre stratégie)

- Weglot structure ses offres autour de **limites de mots traduits + nombre de langues** (ex. free plan < 2 000 mots, 1 langue). weglot.com +1
- Ils mettent aussi en avant un volume de langues élevé (110+). weglot.com

👉 *Ça t'aide à cadrer ton propre pricing (mots, pages, cache hit-rate, sites/clients, SLA, etc.).*

AGENDA NEW FUNCTIONS

2) Mapping “Weglot → nous” (fonction par fonction, et ce que je te conseille de faire dans l’ordre)

Étape 0 — Stabiliser le “core” (anti-boucle, anti-coût)

Objectif : **une fois traduit = plus d’appels API** (ton point critique actuel).

1. **Translation Memory** (cache fort) : clé = `(lang_src, lang_dst, normalized_text, context_hint?) → translated_text`
2. **Dédup batch** : regrouper les textes identiques trouvés sur une page, traduire une fois.
3. **Marquage DOM** (ou hash) : si un nœud est déjà traduit et inchangé → skip.
4. **Invalidation** : si texte source change → retraduire, sinon jamais.

> C’est littéralement l’équivalent “Translation memory + content detection” de Weglot, mais adapté à votre parser. weglot.com

Étape 1 — Règles & exclusions (contrôle + performance)

Objectif : réduire le bruit, traduire uniquement ce qui doit l’être.

- Exclusions par **sélecteurs CSS** (ex : `.no-i18n`, `[data-i18n="off"]`)
- Exclusions par **patterns URL** (`/admin/*`, `/account/*`)
- Exclusions par **types** (scripts, code, JSON inline, etc.)

  Référence côté Weglot : “Exclusions”. weglot.com +1

Étape 2 — Dashboard MVP (vrai “produit”)

Objectif : sortir du mode “script” → mode “plateforme”.

- Liste des pages détectées + % traduit
- Table “strings” (source → cible), filtres (par URL, par statut, par date)
- Edit inline + save → push immédiat

  Weglot met clairement en avant édition via dashboard. weglot.com +1

Étape 3 — Visual Editor (le “wow effect”)

Objectif : le truc qui fait vendre.

- Iframe du site + survol d’éléments traduits + edit en contexte
- Mise à jour temps réel

  Weglot : “Visual editing / live preview”. weglot.com +1

Étape 4 — SEO multilingual (si vous visez le marché “sérieux”)

Objectif : indexation Google propre.

- URLs dédiées `/en/`, `/es/` **ou** `en.domain.com`
- `hreflang` auto sur toutes les pages
- canonical / sitemap cohérent

  Weglot : URLs dédiées + hreflang + explications sitemap/SEO. weglot.com +2

Étape 5 — Import/Export + Collaboration

Objectif : industrialiser.

- Export CSV/JSON + import (versioning simple)
- Rôles (admin/editor/viewer), multi-sites, logs d’audit

  Weglot : import/export + collaboration mentionnée. weglot.com +2

Étape 6 — Options avancées (quand le core est clean)

- Glossary rules avancées (regex, "protected terms", "never translate") weglot.com +1
- Gestion plus robuste du **dynamique** (SPA, lazy content, etc.) support.weglot.c... +1
- Monitoring : quotas, coût, cache hit-rate, "calls avoided", latence, erreurs

3) Ton “Weglot Light” : ce que vous avez déjà vs ce qui manque le plus

Déjà là (d’après ce que tu décris)

- Détection DOM + traduction automatique via Google v2
- Switch de langue basique
- Début de scoping

Le gap le plus urgent

1. **Anti-polling / anti-retranslate** (TM + marquage + invalidation)
2. **Exclusions & règles** (pour éviter de traduire du bruit)
3. **Dashboard MVP** (sinon c’est ingérable à l’échelle multi-sites)

Le gap “produit premium”

Mapping “Weglot
Étape 0 — Stabiliser le “core” (anti-boucle, anti-coût)

AGENDA
Stabiliser le core : architecture cible (simple et robuste)

A) Translation Memory (cache fort)

B) Dédup batch (traduire une fois par page)

C) Marquage DOM / hash (ne plus rescanner-traduire en boucle)

D) Invalidation (retraduit seulement si la source change)

A) Translation Memory (cache fort)

But : éviter l'appel API si on a déjà traduit la même chaîne.

Clé recommandée :

- `lang_src`, `lang_dst`
- `normalized_text` (trim + collapse spaces + normalisation unicode)
- `context_hin` Demander à ChatGPT `h` ou `scope_id` si tu veux éviter des collisions "bizarres")

Key string (stable) :

```
tm:{src}:{dst}:{sha1(normalized_text)}:{sha1(context_hint||"")}
```

Stockage :

- **L1** : mémoire JS (Map) pour la page en cours (ultra rapide)
- **L2** : localStorage/IndexedDB (cache client persistant) si tu veux
- **L3** : DB serveur (PostgreSQL/MariaDB) = source de vérité multi-pages/multi-sessions /multi-sites

> Au début, L3 suffit déjà, mais L1 + L3 = énorme gain perfs.

B) Dédup batch (traduire une fois par page)

But : si 30 nœuds contiennent “Contact”, tu ne fais qu’un appel.

Principe :

1. Lors du scan, tu extrais des “items” `{node, rawText, normText, key}`
2. Tu groupes par `key` (ou par `normText`)
3. Tu construis une liste **unique** à traduire
4. Tu appliques la réponse à **tous** les nœuds du groupe

Résultat : sur une page réelle, tu passes souvent de **1000 strings → 150 strings.**

C) Marquage DOM / hash (ne plus rescanner-traduire en boucle)

C'est ton bug actuel : le parser "continue d'analyser" et déclenche des calls.

But : marquer un nœud comme "déjà traité pour cette langue + cette source".

Marquage recommandé (dataset) :

- `data-i18n-dst="en"`
- `data-i18n-src-hash="<hash(normalized_source)>"`

Règle de skip :

Si `node.dataset.i18nDst === dstLang` ET `node.dataset.i18nSrcHash === hash(normTextActuel)`

→ **skip** (pas de TM lookup, pas d'API, rien)

> Important : le hash doit être basé sur le **texte source** (pas le texte traduit).

D) Invalidation (retraduit seulement si la source change)

Quand la source change ?

- le texte original du nœud change (ex : contenu CMS, JS dynamique)
- ou changement de langue source/destination
- ou changement de règles (glossary, exclusions, etc.)

Comment le détecter simplement :

- tu recalcule `srcHash = hash(normalized_text_source)`
- si `srcHash != node.dataset.i18nSrcHash` → invalide → retraduire

Et côté serveur TM :

- si tu stockes TM par clé `sha1(normalized_text)` : pas besoin "d'invalidation" serveur
- si le texte change, la clé change → nouveau record
- tu peux faire du "cleanup" plus tard via cron (purge des entrées non utilisées)

AUDIT FONCTIONNEL DE WEGLOT

AGENDA AUDIT FONCTIONNEL - WEGLOT

- A. Détection & traduction automatique “always-on”
- B. Dashboard de gestion des traductions (TMS simplifié)
- C. Glossary & Translation Memory (qualité + cohérence + coût)
- D. Exclusions & règles de traduction (contrôle fin)
- E. Language switcher (UX)
- F. Multilingual SEO (gros différenciateur)
- G. Import / Export
- H. Modèle économique (à connaître pour votre stratégie)

1) Ce que Weglot propose (état des lieux, “blocs de fonctionnalités”)

A. Détection & traduction automatique “always-on”

- **Scan / détection de contenu** sur tout le site + traduction initiale (ils parlent de “content detection”, “automatic content detection”). weglot.com +1
- **Traduction continue** : dès qu’un contenu change/une page apparaît, ça repart en détection + traduction. weglot.com +1
- Gestion des **contenus dynamiques** (avec des guides dédiés). support.weglot.c... +1

👉 *Votre “Weglot Light” a déjà une partie de ça (scan + traduction), mais pas encore l’équivalent “toujours synchronisé” proprement (et sans re-calls).*

B. Dashboard de gestion des traductions (TMS simplifié)

- Interface pour **voir/éditer les traductions**, généralement organisée par URL / langue. weglot.com +1
- **Visual Editor** : édition “in context” dans une prévisualisation live du site. weglot.com +1
- **Gestion de tâches / workflow** (le marketing Weglot pousse “auto-pilot”, prioriser les pages visibles, etc.). weglot.com +1
- **Collaboration** (multi-utilisateurs/équipe) selon intégrations/plans (souvent mentionné dans leurs contenus comparatifs). brixtemplates.com +1

👉 *Vous avez sûrement : stockage + affichage minimal. Il manque : visual editor + workflow + permissions + outils de tri/filtre.*

C. Glossary & Translation Memory (qualité + cohérence + coût)

- **Glossaire / règles** : protéger des termes (“ne pas traduire X”, “toujours traduire Y par Z”). weglot.com +2
- **Translation Memory** : réutiliser les traductions existantes pour éviter retraductions. weglot.com +1

👉 *C’est LE levier direct pour ton problème actuel (des milliers d’appels inutiles) : TM + cache + dédup + invalidation propre.*

D. Exclusions & règles de traduction (contrôle fin)

- **Exclusions** (pages, blocs, sélecteurs, patterns) pour ne pas traduire certaines zones. weglot.com +1
- (En pratique Weglot supporte beaucoup de “rules”, par sélecteurs/URLs/éléments — ils l’annoncent surtout via “Exclusions”). weglot.com

👉 *Vous avez déjà commencé à parler de “scopes” côté parser : Weglot formalise ça en règles éditables.*

E. Language switcher (UX)

- **Sélecteur de langue** personnalisable (flags/labels/design) + intégration selon CMS. weglot.com +1

👉 *Côté UI vous êtes en plein dedans : dropdown propre + langue courante sur la même ligne, sans “truc moche”.*

F. Multilingual SEO (gros différenciateur)

- **URLs dédiées par langue** (subdomain ou subdirectory) pour indexation. weglot.com +1
- **hreflang auto**, + gestion SEO associée. weglot.com +2
- Weglot insiste que ce n'est pas "juste du JS" : ils mettent en avant l'approche "server-side / embedded in source" quand c'est configuré en subdomain/subdirectory. support.weglot.c... +1
- **Sitemap & SEO meta** (ils expliquent la logique sitemap/hreflang). support.weglot.c... +1

👉 *Si vous voulez être "Weglot-like" commercialement : le SEO multi-langues est un module à part entière, pas juste "on traduit le DOM".*

G. Import / Export

- **Export/import** des traductions pour bosser hors dashboard puis réimporter. weglot.com

👉 *Facile à livrer en V1.5 : CSV/JSON + API endpoints, ça fait très "pro".*

H. Modèle économique (à connaître pour votre stratégie)

- Weglot structure ses offres autour de **limites de mots traduits + nombre de langues** (ex. free plan < 2 000 mots, 1 langue). weglot.com +1
- Ils mettent aussi en avant un volume de langues élevé (110+). weglot.com

👉 *Ça t'aide à cadrer ton propre pricing (mots, pages, cache hit-rate, sites/clients, SLA, etc.).*

MAPPING « WEGLOT » → IDEO-LAB

AGENDA MAPPING WEGLOT -> IDEOLAB

- Étape 0 — Stabiliser le “core” (anti-boucle, anti-coût)
- Étape 1 — Règles & exclusions (contrôle + performance)
- Étape 2 — Dashboard MVP (vrai “produit”)
- Étape 3 — Visual Editor (le “wow effect”)
- Étape 4 — SEO multilingual (si vous visez le marché “sérieux”)
- Étape 5 — Import/Export + Collaboration
- Étape 6 — Options avancées (quand le core est clean)

Étape 0 — Stabiliser le “core” (anti-boucle, anti-coût)

Objectif : **une fois traduit = plus d’appels API** (ton point critique actuel).

1. **Translation Memory** (cache fort) : clé = `(lang_src, lang_dst, normalized_text, context_hint?) → translated_text`
2. **Dédup batch** : regrouper les textes identiques trouvés sur une page, traduire une fois.
3. **Marquage DOM** (ou hash) : si un nœud est déjà traduit et inchangé → skip.
4. **Invalidation** : si texte source change → retraduire, sinon jamais.

> C’est littéralement l’équivalent “Translation memory + content detection” de Weglot, mais adapté à votre parser. weglot.com

Étape 1 — Règles & exclusions (contrôle + performance)

Objectif : réduire le bruit, traduire uniquement ce qui doit l'être.

- Exclusions par **sélecteurs CSS** (ex : `.no-i18n`, `[data-i18n="off"]`)
- Exclusions par **patterns URL** (`/admin/*`, `/account/*`)
- Exclusions par **types** (scripts, code, JSON inline, etc.)

  Référence côté Weglot : "Exclusions". weglot.com +1

Étape 2 — Dashboard MVP (vrai “produit”)

Objectif : sortir du mode “script” → mode “plateforme”.

- Liste des pages détectées + % traduit
- Table “strings” (source → cible), filtres (par URL, par statut, par date)
- Edit inline + save → push immédiat

  Weglot met clairement en avant édition via dashboard. weglot.com +1

Étape 3 — Visual Editor (le “wow effect”)

Objectif : le truc qui fait vendre.

- Iframe du site + survol d’éléments traduits + edit en contexte
- Mise à jour temps réel

  Weglot : “Visual editing / live preview”. weglot.com +1

Étape 4 — SEO multilingual (si vous visez le marché “sérieux”)

Objectif : indexation Google propre.

- URLs dédiées `/en/`, `/es/` **ou** `en.domain.com`
- `hreflang` auto sur toutes les pages
- canonical / sitemap cohérent

  Weglot : URLs dédiées + hreflang + explications sitemap/SEO. weglot.com +2

Étape 5 — Import/Export + Collaboration

Objectif : industrialiser.

- Export CSV/JSON + import (versioning simple)
- Rôles (admin/editor/viewer), multi-sites, logs d'audit

  Weglot : import/export + collaboration mentionnée. weglot.com +2

Étape 6 — Options avancées (quand le core est clean)

- Glossary rules avancées (regex, “protected terms”, “never translate”) weglot.com +1
- Gestion plus robuste du **dynamique** (SPA, lazy content, etc.) support.weglot.c… +1
- Monitoring : quotas, coût, cache hit-rate, “calls avoided”, latence, erreurs

Implémentation “pratique”

IMPLEMENTATION AGENDA

- 0) Stabiliser le core : architecture cible (simple et robuste)
  - A) Translation Memory (cache fort)
  - B) Dédup batch (traduire une fois par page)
  - C) Marquage DOM / hash (ne plus rescanner-traduire en boucle)
  - D) Invalidation (retraduit seulement si la source change)
- 1) Implémentation “pratique” : le flux exact à faire dans ton parser JS
- 2) Points qui évitent 90% des “milliards de calls” (les pièges classiques)
- 3) Backend (Django) : schéma de table TM (simple)
- 4) Roadmap “mini” (livrable en 2 itérations)

0) Stabiliser le core : architecture cible (simple et robuste)

Les 4 piliers (et comment les coder)

A) Translation Memory (cache fort)

But : éviter l'appel API si on a déjà traduit la même chaîne.

Clé recommandée :

- `lang_src`, `lang_dst`
- `normalized_text` (trim + collapse spaces + normalisation unicode)
- `context_hint` **optionnel** (ex: `url_path` ou `scope_id` si tu veux éviter des collisions "bizarres")

Key string (stable) :

`tm:{src}:{dst}:{sha1(normalized_text)}:{sha1(context_hint||"")}`

Stockage :

- **L1** : mémoire JS (Map) pour la page en cours (ultra rapide)
- **L2** : localStorage/IndexedDB (cache client persistant) si tu veux
- **L3** : DB serveur (PostgreSQL/MariaDB) = source de vérité multi-pages/multi-sessions /multi-sites

B) Dédup batch (traduire une fois par page)

But : si 30 nœuds contiennent "Contact", tu ne fais **qu'un** appel.

Principe :

1. Lors du scan, tu extrais des "items" `{node, rawText, normText, key}`
2. Tu groupes par `key` (ou par `normText`)
3. Tu construis une liste **unique** à traduire
4. Tu appliques la réponse à **tous** les nœuds du groupe

Résultat : sur une page réelle, tu passes souvent de **1000 strings → 150 strings**.

C) Marquage DOM / hash (ne plus rescanner-traduire en boucle)

C'est ton bug actuel : le parser "continue d'analyser" et déclenche des calls.

But : marquer un nœud comme "déjà traité pour cette langue + cette source".

Marquage recommandé (dataset) :

- `data-i18n-dst="en"`
- `data-i18n-src-hash="<hash(normalized_source)>"`

Règle de skip :

Si `node.dataset.i18nDst === dstLang` ET `le.dataset.i18nSrcHash === hash(normTextActuel)`

D) Invalidation (retraduit seulement si la source change)

Quand la source change ?

- le texte original du nœud change (ex : contenu CMS, JS dynamique)
- ou changement de langue source/destination
- ou changement de règles (glossary, exclusions, etc.)

Comment le détecter simplement :

- tu recalcule `srcHash = hash(normalized_text_source)`
- si `srcHash != node.dataset.i18nSrcHash` → invalide → retraduire

Et côté serveur TM :

- si tu stockes TM par clé `sha1(normalized_text)` : pas besoin "d'invalidation" serveur
- si le texte change, la clé change → nouveau record
- tu peux faire du "cleanup" plus tard via cron (purge des entrées non utilisées)

1) Implémentation “pratique” : le flux exact à faire dans ton parser JS

Pipeline recommandé (ordre exact)

1. **Scan** DOM → récupère nodes texte “traduisibles”
2. Pour chaque node :
   - `raw = node.textContent`
   - `norm = normalize(raw)`
   - si vide / trop court / numeric only / excluded → skip
   - `srcHash = hash(norm)`
   - si node déjà marqué et hash identique → skip
   - push dans `items`
3. **Dédup** : group by `key = (src,dst,srcHash,contextHint?)`
4. **TM lookup** :
   - d’abord L1 Map (in-memory)
   - puis backend `/tm/batch_get`
5. **Traduire uniquement les “miss”** via `/translate/batch`
6. **Écrire TM** (backend) `/tm/batch_set` (ou inclus dans translate response)
7. **Appliquer aux nodes**
8. **Marquer nodes** (dst + srcHash) pour bloquer toute boucle
9. **Stopper** (et surtout ne pas repoll inutilement)

2) Points qui évitent 90% des “milliards de calls” (les pièges classiques)

Piège #1 : Observer qui retraduit ce qu’il vient de modifier

Si tu utilises `MutationObserver`, quand tu modifies `textContent`, il déclenche un nouvel event → boucle.

Solution :

- pendant l’application des traductions : `this._isApplying = true`
- observer callback : si `this._isApplying` → ignore
- puis `this._isApplying = false`

Piège #2 : Traduire le texte déjà traduit

Si ton scan prend `node.textContent` après traduction, tu perds la source.

Solution (2 options) :

- stocker la source dans `data-i18n-src="..."` (attention taille)
- ou mieux : stocker seulement `data-i18n-src-hash` + avoir TM sur le texte source (via DB)

Piège #3 : Normalisation incohérente

Si `normalize()` n’est pas stable, tu “rates” le TM.

Normalize minimal recommandé :

- trim
- collapse whitespace
- normalisation unicode (NFKC)
- ne pas lower-case (selon langues ça pe[illegible]asser)

3) Backend (Django) : schéma de table TM (simple)

Table `translation_memory`

- `site_id` (ou projet/client)
- `src_lang`, `dst_lang`
- `src_hash` (sha1 du normalized_text) indexé
- `src_text` (optionnel, utile debug)
- `dst_text`
- `context_hash` (optionnel)
- `updated_at`, `created_at`
- compteur `hit_count` (optionnel)
- `last_seen_at` (optionnel)

Endpoints

- `POST /api/i18n/tm/batch_get` → `{keys:[...]}` → `{found:{key:translation}}`
- `POST /api/i18n/translate/batch` → `{texts:[...], src, dst}` → `{translations:[...]}`
    - et au retour, tu peux **écrire TM** directement.

4) Roadmap “mini” (livrable en 2 itérations)

Itération 1 (impact immédiat)

- L1 Map + DB TM
- Dédup batch
- DOM marking + skip
- Observer “anti-loop”

✅ Résultat attendu : tu passes de “milliers de calls” à **quasi 0** après premier passage.

Itération 2 (propreté produit)

- context_hint (url/scope)
- métriques : `calls_saved`, `tm_hit_rate`, `strings_unique`
- purge cron des entrées jamais revues

TRANSLATION DATA MODEL

```python
class TranslationUnit(models.Model):
    site = models.CharField(max_length=64, db_index=True)

    src_lang = models.CharField(max_length=12, db_index=True)
    dst_lang = models.CharField(max_length=12, db_index=True)

    key_hash = models.CharField(max_length=64, db_index=True, blank=True, default
    src_text_norm = models.TextField()
    ctx_fingerprint = models.CharField(max_length=120, db_index=True, blank=True,

    dst_text = models.TextField(blank=True, default="")

    engine = models.CharField(max_length=32, blank=True, default="")
    quality = models.CharField(max_length=16, blank=True, default="auto")  # autc

    hit_count = models.PositiveIntegerField(default=0)
    last_seen_at = models.DateTimeField(null=True, blank=True)

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    meta = models.JSONField(blank=True, default=dict)

    def __str__(self):
        return f"{self.site} {self.src_lang}->{self.dst_lang} {self.key_hash or s
```

```python
class ExclusionRule(models.Model):
    """
    Étape 1 : Règles dynamiques pour ignorer du contenu.
    """
    project = models.ForeignKey("TranslationProject", on_delete=models.CASCADE, rel

    RULE_TYPES = [
        ('css', 'CSS Selector'),  # ex: .notranslate, #menu
        ('url', 'URL Pattern'),   # ex: /admin/*
        ('block', 'Block ID'),    # ex: div#footer
    ]
    rule_type = models.CharField(max_length=10, choices=RULE_TYPES, default='css')
    pattern = models.CharField(max_length=255)
    is_active = models.BooleanField(default=True)

    def __str__(self):
        return f"[{self.rule_type}] {self.pattern}"
```

```python
class TranslationProject(models.Model):
    name = models.CharField(max_length=80)
    slug = models.SlugField(max_length=80)
    api_key = models.CharField(max_length=64)
    base_url = models.URLField()

    default_language = models.CharField(max_length=10, default="fr")
    enabled_languages = models.JSONField(default=list)
    # ["en", "es", "de"]

    plan = models.CharField(max_length=32, default="pro")
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

ALREADY APPLIED ON IDEOLAB

AGENDA CODE APPLIED AND INSTALLED

- A) Translation Memory (cache fort) : **OK**
- B) Dédup batch (traduire une fois par page) : NON
- C) Marquage DOM / hash (ne plus rescanner-traduire en boucle) : NON
- D) Invalidation (retraduire seulement si la source change) : NON

0) Stabiliser le core — vérification “plan initial” vs code actuel

A) Translation Memory (cache fort)

✅ Déjà couvert (dans ton code actuel)

L3 (DB serveur) : OUI

- Modèle `TranslationUnit` = TM serveur (clé `key_hash`, `src_text_norm`, `ctx_fingerprint`, `dst_text`, etc.). models
- Endpoint `batch_translate` fait : nettoyage → hits par `key_hash` → traduit seulement les “missing” → stocke en TM via `upsert_like`.

L1/L2 (cache côté JS) : OUI

- Ton JS instancie `this.l1 = new LruCache(...)` et `this.l2 = new StorageCache(...)`. (Donc L1+L2 existent déjà.) general_init_translation

➡️ **Conclusion A : déjà OK** (le TM “Weglot-like” est en place).

CONTEXTE VERTEX AI JAN 16 2026
STATUT EN COURS

1. Le Frontend (`ideolab-i18n.js` + `general_init_translation.html`)

- **Initialisation :** Le script démarre, appelle `init()`.
- **Config Distante :** Il tape `GET /api/i18n/v1/config` pour récupérer les règles (CSS/URL) gérées par le backend.
- **Protection :** Il utilise `RulesEngine` pour ignorer les éléments AVANT de scanner (via `isIgnoredElement` qui checke `this.dynamicCss`).
- **Core Logic :** Scan DOM -> Fingerprint -> Vérification Cache (L1/L2) -> Batch API -> Application avec `MutationObserver` sécurisé (`_inApplyGuard`).

2. Le Backend (`views.py` , `urls.py`)

- **Endpoint Config :** `project_config` reçoit la clé API, trouve le `TranslationProject`, et renvoie les `ExclusionRule` actives (CSS/URL).
- **Endpoint Batch :** `batch_translate` reçoit les fragments, normalise, vérifie la TM (`TranslationUnit`), et appelle Google si nécessaire (via `gcp_translate.py`).

3. La Data (`models.py` - Source de Vérité)

Note : Le fichier contient plusieurs définitions historiques, mais je me focalise sur celles qui font tourner le moteur actuel :

- `TranslationProject` : La clé de voûte (API Key, Langues, Settings).
- `ExclusionRule` : Les règles dynamiques (ex: `.no-translate`).
- `TranslationUnit` : La mémoire de traduction (Source Text + Hash + Traduction + Context).

Statut du Projet

Étape PDF	État Code	Verdict
0. Core (TM + Anti-boucle)	✅ Code en place	Le moteur traduit, déduplique et cache.
1. Règles & Config	✅ Code en place	Le JS charge la config du serveur. Le Backend sert les règles.
2. Dashboard MVP	⏳ **À FAIRE**	Visualiser et éditer les traductions.

Exporter vers Sheets

La Suite : Étape 2 (Le Dashboard MVP)

Selon ton PDF (Page 36), l'objectif maintenant est de : **"Sortir du mode script › mode plateforme"**.

Nous avons des données qui rentrent dans `TranslationUnit`, mais c'est une "boîte noire". Il faut maintenant une interface pour voir ces traductions et les corriger.

Proposition pour démarrer l'Étape 2 : On ne va pas faire du React complexe tout de suite. On va faire une **vue Django "Liste des Traductions"** simple mais puissante, filtrable par langue et par statut.

Veux-tu que je génère :

1. Le code **HTML/Template Django** pour un tableau de bord "Weglot-like" (Liste des strings, Recherche, Édition rapide) ?
2. La **Vue Django** associée pour afficher les `TranslationUnit` paginées ?

Bilan de l'Étape 1 : Règles & Exclusions

Fonctionnalité requise (PDF p.35)	État dans ton Code	Preuve
Exclusion par Sélecteur CSS	✅ **OK**	**Backend :** `ExclusionRule` gère le type `css`. **Front :** `RulesEngine` filtre via `isIgnoredElement`.
Exclusion par URL Pattern	✅ **OK**	**Backend :** `ExclusionRule` gère le type `url`. **Front :** `init()` vérifie `isUrlIgnored` et stoppe tout si match.
Chargement Dynamique	✅ **OK**	**API :** `/api/i18n/v1/config` envoie les règles au chargement.

Exporter vers Sheets

PERSONAL TO DO - 2026

- Ajouter une progress bar discrète dans la page en cours de traduction
- Mieux Packager le petit script de démarrage du weglot like (***base HTML***)
- Le user doit pouvoir selectionner ou exclure certains DOM Elements :
  - Bouttons, Titres, Url, Nav (***par Categorie***)
- Offrir le choix du Client d'utiliser son propre Traducteur
- Pré-traduire des urls complète à l'avance
- Connaitre en temps réel la consommation de traductions (***Translate Engine***)
- Offrir le choix du compte « Engine translation », option and API Key code
- Améliorer les icones de selecteur de langues (fonction & css)
  - Prévoir comme weglot un sélecteur « flottant » en bas à droite de la page
  - Ce selecteur de langue sera généré par défaut
  - Ajouter un Premier DashBoard qui permettra de piloter les traduction et langues

ERREUR ET PROBLEMES URGENTS

ERREUR ET PROBLEMES URGENT

- LATENCE TRES FORTE POUR AFFICHER UNE PAGE DEJA TRADUITE
- MANQUE UNE PROGRESS BAR PENDANT LE TRADUCTION
- LANGAGE SELECTEUR A REVOIR

PERFORMANCES ISSUES

🥇 PRIORITÉ 1 — Chunking (batch phrases)

Objectif : réduire **drastiquement** la latence perçue et réelle **sans changer le front**.

🎯 Principe (validé)

- Le JS **ne change pas** : il envoie toujours une liste de chaînes.
- Le **backend regroupe** ces chaînes par paquets (chunks).
- Chaque chunk est traduit **en un seul appel** à Google.
- Le résultat est **re-découpé** et redistribué comme si chaque phrase avait été traduite individuellement.

👉 Transparence totale pour le front, le cache, la dédup, la progress bar.

🧱 Où l'implémenter (verrouillé)

- ❌ pas dans le JS
- ❌ pas dans les views
- ❌ pas dans `engine.py` (orchestration)
- ✅ **dans** `gcp_translate.py` (point d'appel externe)

C'est **le bon niveau d'abstraction**.

🥇 PRIORITÉ 1 — Chunking (batch phrases)

- énorme gain
- peu de complexité
- peu de risques
- effet immédiat sur la progress bar

🥈 PRIORITÉ 2 — Cache / TranslationUnit (déjà en place)

- déjà bien
- à améliorer plus tard

🥉 PRIORITÉ 3 — Parallélisme

- utile à grande échelle
- à activer avec quotas clairs
- pas nécessaire tout de suite

✅ Suite (UNE seule direction, la bonne) : pré-traduire / préchauffer

Tu l'as dit toi-même : *"il faut tout prétraduire"* — **oui**.

Ce qu'on met en place

Un **cron Django / management command** qui :

1. prend une liste d'URLs (ton site)
2. récupère les fragments (soit via ton parser déjà existant, soit via une "API scan")
3. appelle ton backend `batch_translate`
4. remplit `TranslationUnit` / cache L2
5. résultat : au premier visiteur → **presque tout est hit**, donc instantané.

👉 C'est EXACTEMENT le modèle Weglot.

PRE WARMING DES TRADUCTIONS

1) Management command “prewarm” (SITEMAP → pages → cache rempli)

Crée : `i18n_engine/management/commands/i18n_prewarm.py`

python

Copier le code

```python
# i18n_engine/management/commands/i18n_prewarm.py
from __future__ import annotations

import sys
import time
import xml.etree.ElementTree as ET
from urllib.parse import urljoin

import requests
from django.core.management.base import BaseCommand
from django.conf import settings


def _fetch(url: str, timeout: int = 20) -> str:
    r = requests.get(url, timeout=timeout, headers={"User-Agent": "IdeoLab-i18
    r.raise_for_status()
    return r.text
```

2) Ce que ça te donne (effet immédiat)

- Tu lances le prewarm **une seule fois** (ou nightly).
- Tes pages "lourdes" sont traduites **avant** l'utilisateur.
- Ensuite, en navigation réelle : tu dois voir **L2 hits énormes** et **remote quasi nul.**

---

3) Commandes à exécuter

Sur ton Django :

bash Copier le code

```
pip install playwright
python -m playwright install chromium
python manage.py i18n_prewarm --base http://127.0.0.1:8000 --sitemap /sitemap.
```

TO DO LIST 2026

PERSONAL TO DO - 2026

- Ajouter une progress bar discrète dans la page en cours de traduction
- Mieux Packager le petit script de démarrage du weglot like (***base HTML***)
- Le user doit pouvoir selectionner ou exclure certains DOM Elements :
  - Bouttons, Titres, Url, Nav (***par Categorie***)
- Offrir le choix du Client d'utiliser son propre Traducteur
- Pré-traduire des urls complète à l'avance
- Connaitre en temps réel la consommation de traductions (***Translate Engine***)
- Offrir le choix du compte « Engine translation », option and API Key code
- Améliorer les icones de selecteur de langues (fonction & css)
  - Prévoir comme weglot un sélecteur « flottant » en bas à droite de la page
  - Ce selecteur de langue sera généré par défaut
  - Ajouter un Premier DashBoard qui permettra de piloter les traduction et langues

TODO - OPTIMISATION - STAT

- Il faudrait connaitre l'état des traductions par page HTML
- Il faudrait aussi pre-traduire des pages HTML (par Url) à la demande
- Il faudrait aussi connaitre la liste des pages (url) qui restent à traduire
- Enrichir et améliorer les règles d'exclusions, trop simplistes on risque !!

ROADMAP PHASE 2

AGENDA PHASE 2 ET 3

- **2.1 ◆ Détection & tracking automatique des pages vues**
- **2.2 ◆ Priorisation intelligente des URLs**
- **2.3 ◆ Pre-warm DOM segments (le vrai cœur Weglot)**
- **3.1 ◆ Résolution instantanée (zéro latence)**
- **3.2 ◆ Sélecteur de langue Weglot-like**
- **4.1 ◆ Dashboard client**
- **4.2 ◆ Sécurité & clés**

PHASE 2 — CORE WEGLOT (prochaine étape logique)

2.1 🔹 Détection & tracking automatique des pages vues

👉 Comme Weglot :

- le JS intercepte les pages visitées
- envoie `/current-url` au backend
- si URL inconnue → ajout auto dans `WeglotURL`

💡 Bénéfice :

- plus besoin de tout saisir à la main
- découverte naturelle du site

📌 Technique :

- petit endpoint `POST /api/weglot/v1/url/seen`
- payload `{ url, referrer }`
- `get_or_create(WeglotURL)`

2.2 🔹 Priorisation intelligente des URLs

Weglot ne traduit pas tout pareil.

Ajouter dans `WeglotURL` :

- `priority` (int)
- `hit_count` (vu par le JS)

Le cron :

- traite d'abord `priority DESC, hit_count DESC`
- rafraîchit les pages "chaudes"

2.3 🔹 Pre-warm DOM segments (le vrai cœur Weglot)

Actuellement tu fais :

- title / h1 / meta → parfait

Étape suivante :

- parser HTML serveur
- extraire :
  - text nodes visibles
  - buttons
  - labels
  - placeholders

⚠️ MAIS :

- **limité**
- **batché**
- **dédupliqué**

👉 exactement ce que tu fais déjà côté frontend, mais **en avance**.

PHASE 3 — FRONTEND MAGIC (UX Weglot)

3.1 🔹 Résolution instantanée (zéro latence)

Le JS doit :

- demander `/resolve`
- recevoir **déjà traduit**
- ne jamais appeler Google en live (sauf fallback)

Tu y es presque.

3.2 🔹 Sélecteur de langue Weglot-like

- dropdown simple
- persistance (cookie / localStorage)
- redirection propre

PHASE 4 — SaaS / Produit

4.1 🔹 Dashboard client

- liste des pages traduites
- erreurs
- boutons : retraduire / exclure
- stats de consommation

4.2 🔹 Sécurité & clés

- `publishable_key` (frontend)
- `secret_key` (dashboard / cron)
- rate limit
- quotas

VERIFICATIONS

1) Vérification immédiate (à faire maintenant)

Avant d'ajouter des features, on valide que le core est **vraiment** stable.

1.1 Mesures à regarder (dans la console / overlay)

- **Scan 1** : `remote_calls > 0` (normal)
- **Scan 2 / rescan MutationObserver** : `remote_calls ≈ 0` (ou très faible)
- **uniqueSent / batchSize** : doit montrer la **dédup** (ex : 150 / 900)

Si tu ne l'as pas, je te conseille d'ajouter 3 logs (sans refactor) :

- `misses_count`
- `unique_sent_count`
- `applied_skip_count`

2) Itération 2 (propreté "produit") — *invalidation + robustesse dynamique*

C'est la prochaine brique indispensable.

2.1 Invalidation (D) — retraduire seulement si la source change

Aujourd'hui tu captures la source "une fois". C'est bien… mais si un texte est mis à jour (CMS / JS), tu dois :

- détecter que la source a changé
- mettre à jour `srcRaw/srcNorm/srcHash`
- invalider `applied[to]`
- retraduire

Concrètement :

Dans `_collectTextFragments()` et `_collectAttrFragments()` :

- tu compares le "texte courant" avec le `st.srcRaw` initial
- si différent (après normalisation), alors :
  - `st.srcRaw = rawCurrent`
  - `st.srcNorm = normCurrent`
  - `st.srcHash = fnv1a64(normCurrent)`
  - `delete st.applied[this.cfg.to]` (ou reset `applied = {}`)

Même chose pour les attributs.

👉 C'est LE point qui te permet de gérer correctement les pages dynamiques **sans jamais retomber dans la boucle.**

2.2 MutationObserver “smart”

Actuellement, tu rescannes “large”. Prochaine amélioration :

- sur mutation, tu ne rescannes que le **sous-arbre** impacté (target / addedNodes)
- - debounce (tu as déjà un debounce, mais on peut le rendre plus sélectif)

Résultat : moins de CPU, moins de latence, meilleure UX.

3) Qualité traduction (Weglot-like) — Glossary / protected terms

Une fois le coût et les boucles maîtrisés, on améliore la qualité :

- “ne jamais traduire” certaines marques / acronymes
- “toujours traduire X par Y”
- règles simples par langue

Tu as déjà `rules.py` côté backend et des hooks front, donc c’est une itération très rentable.

4) Roadmap “produit” (si tu veux passer multi-sites proprement)

- Dashboard MVP : pages détectées, strings, statut, édition
- Export/import CSV/JSON
- SEO (subdir / hreflang) plus tard

1) Implémentation “pratique” : le flux exact à faire dans ton parser JS

Pipeline recommandé (ordre exact)

1. **Scan** DOM → récupère nodes texte “traduisibles”
2. Pour chaque node :
   - `raw = node.textContent`
   - `norm = normalize(raw)`
   - si vide / trop court / numeric only / excluded → skip
   - `srcHash = hash(norm)`
   - si node déjà marqué et hash identique → skip
   - push dans `items`
3. **Dédup** : group by `key = (src,dst,srcHash,contextHint?)`
4. **TM lookup** :
   - d’abord L1 Map (in-memory)
   - puis backend `/tm/batch_get`
5. **Traduire uniquement les “miss”** via `/translate/batch`
6. **Écrire TM** (backend) `/tm/batch_set` (ou inclus dans translate response)
7. **Appliquer aux nodes**
8. **Marquer nodes** (dst + srcHash) pour bloquer toute boucle
9. **Stopper** (et surtout ne pas repoll inutilement)

Piège #1 : Observer qui retraduit ce qu'il vient de modifier

Si tu utilises `MutationObserver`, quand tu modifies `textContent`, il déclenche un nouvel event → boucle.

Solution :

- pendant l'application des traductions : `this._isApplying = true`
- observer callback : si `this._isApplying` → ignore
- puis `this._isApplying = false`

Piège #2 : Traduire le texte déjà traduit

Si ton scan prend `node.textContent` après traduction, tu perds la source.

Solution (2 options) :

- stocker la source dans `data-i18n-src="..."` (attention taille)
- ou mieux : stocker seulement `data-i18n-src-hash` + avoir TM sur le texte source (via DB)

Piège #3 : Normalisation incohérente

Si `normalize()` n'est pas stable, tu "rates" le TM.

Normalize minimal recommandé :

- trim
- collapse whitespace
- normalisation unicode (NFKC)
- ne pas lower-case (selon langues ça peut casser)

SOME IDEAS ABOUT OPTIMIZATIONS

1) “Traduire toute la page d’un coup” : oui on peut tester, mais avec prudence

Option A — Whole page “HTML translation” (expérience simple)

Tu construis une string HTML, tu l’envoies à Google en **mimeType** = **text/html**, tu reçois du HTML traduit, et tu remplaces le contenu.

Avantages

- 1 appel (ou très peu) → énorme gain sur API/roundtrips.
- Réduction massive du temps passé à “redispatch” des milliers de fragments.

Risques / pièges

- Les traducteurs peuvent “toucher” des morceaux sensibles (attributs, URLs, `data-*`, bouts de JS inline, etc.).
- Remplacer du HTML peut casser des listeners / composants (React/Vue), des états, des formulaires.
- Les scripts et certains nœuds ne doivent jamais être traduits.

👉 Donc c’est testable **avec un flag**, mais je le mettrais d’abord sur des pages “simples” (articles statiques, docs) pour mesurer.

Option B — Whole page "text-only map" (plus robuste)

Tu **n'envoies pas le HTML** au traducteur : tu extrais tous les *text nodes* (et éventuellement certains attributs comme `title`, `placeholder`, `aria-label`), tu fabriques **un seul payload** type :

- Une liste ordonnée de segments `["Hello", "My account", "Read more", ...]`
- Tu traduis ça en batch
- Tu réinjectes les traductions dans les mêmes nœuds

Avantages

- Zéro risque que le moteur "casse" la structure HTML.
- Toujours 1 (ou très peu) appels API si tu batches bien.
- Compatible avec des pages dynamiques (tu remplaces juste du texte, pas la structure).

Inconvénient

- Il te faut un mapping stable (node references / ids), mais c'est faisable.

➡️ En vrai, **Option B** est souvent le "sweet spot" pour un weglot-like sérieux.

2) Ton problème clé : “même quand c’est déjà traduit, ça lag”

Ça, c’est presque toujours **un symptôme JS/DOM**, typiquement :

- MutationObserver qui retrigger en boucle (ou presque)
- Trop de “scan complet DOM” à chaque micro-changement
- Normalisation/dédoublonnage coûteux à chaque load
- Trop d’accès localStorage / JSON parse/stringify
- Réinjection qui provoque de nouvelles mutations, donc re-scan
- Absence de “fast path” : si cache hit → tu devrais *ne presque rien faire*

Objectif : quand la page est “déjà traduite”, ton moteur doit faire :

1. détecter langue
2. vérifier “page cache hit”
3. appliquer en **O(1)** ou **O(n_text_nodes)** *sans* re-parser/normaliser/segmenter

3) Oui : je ferais exactement ton flag (mais je testerais 2 flags)

Je te propose :

- `WEGLOT_WHOLE_PAGE_MODE = "off" | "html" | "textmap"`
- `WEGLOT_PAGE_CACHE = True/False` (cache HTML/page-level)

Comme ça tu peux comparer proprement :

- **Mode actuel** (fragment/DOM)
- **Textmap batch** (souvent meilleur)
- **HTML full** (peut être très rapide mais plus risqué)

4) Le test le plus “révélateur” (et rapide à implémenter)

Expérience #1 — “Textmap batch” (recommandée en premier)

But : prouver que la lenteur vient du “fragment dispatch”, pas de Google.

Pipeline :

1. Walk DOM → récupère *uniquement* les text nodes traduisibles
2. Concat en une liste, chunk si besoin (limite API)
3. Un appel batch (ou quelques uns)
4. Réinjecte dans les nœuds

Mesure 3 timings :

- `T_scan_dom`
- `T_translate_api`
- `T_inject_dom`

Si `T_scan_dom + T_inject_dom` est énorme → tu sais où optimiser.

Expérience #2 — Cache page-level (game changer sur “déjà traduit”)

Dès que tu as une traduction stable d’une page :

- tu stockes **le résultat complet** (au moins les segments traduits, idéalement une “snapshot” page-level)
- la 2e visite : tu ne rappelles pas l’API, tu ne recalcules presque rien

C’est probablement **le secret principal** derrière ton “Weglot quasi instant”.

5) “Repenser tout cela” : la direction architecture la plus efficace

Si tu veux te rapprocher de Weglot en perf perçue :

1. **Cache page-level par URL+lang** (backend + éventuellement disque/Redis)
2. **Pré-traduction** (sitemap crawler) pour remplir le cache
3. **Frontend ultra-léger** :
    - au chargement : si “page cache exists” → apply direct (ou même servir la page déjà traduite côté serveur)
    - sinon : traduction + stockage + apply

👉 Le “graal” : servir directement **la version traduite** depuis le backend (ou un edge cache)

→ le JS ne fait presque plus rien.