Validation du moteur Python v2

PROJET UNMINIFY & JS ANALYZER

IDEO-LAB

JANUARY 25TH 2026

CDCF v1 (résumé)

1. **Beautify** HTML/CSS/JS
2. **Unminify + rapport** (structure, endpoints, risques, features détectées)
3. **JS reverse toolkit** :
   - "Map des modules" / "Top fonctions"
   - "Network map"
   - "Suspicious patterns"
   - "Strings & constants extractor"
4. Export : fichiers + zip + "rapport HTML"

AF v1 (modules)

- `unminify.core` : pipeline
- `unminify.parsers` : html/css/js
- `unminify.analyzers` : js_ast, iocs, endpoints, callgraph
- `unminify.renderers` : rapport HTML
- `unminify.tasks` : Celery
- `unminify.ui` : pages + modals (style Ideo-Lab)

Étape 0 — point de départ (ce que je considère comme figé)

- UI Django existante ✅
- Modèles `UnminifyJob / UnminifyFile` ✅
- Workflow Job / Progress / Report ✅
- CDCF + AF validés ✅

👉 **Seule chose à remplacer :**

🔴 le moteur Node appelé dans `tasks.py`

🎯 Étape 1 — Définir le nouveau moteur Python

Livrable de cette étape :

- 📄 **Spécification courte** du moteur Python :
  - Entrées exactes
  - Sorties exactes
  - Fichiers générés (`progress.json` , `report.html` , artefacts)
  - Compatibilité 100 % avec l'UI actuelle

⚠️ **Aucun code encore**, uniquement la spec technique finale.

Une fois validée → on code.

SPEC — Moteur Python v2 (sans Node/NPM)

AGENDA NOUVEAU ENGINE V2

- 1) Objectif
- 2) Contrat d’entrée
- 3) Contrat de sortie (100% compatible UI)
- 4) Pipeline v2 (stages)
- 5) Exigences non-fonctionnelles
- 6) Impacts code (pour l’étape 2, pas maintenant)

1) Objectif

Remplacer l'exécution Node de `tasks.py` par un **pipeline 100% Python**, tout en :

- conservant le workflow "Job → Progress → Report → Artefacts"
- améliorant la qualité (lisibilité) + ajoutant des analyses utiles (reverse JS "safe")
- gardant une sortie HTML "report.html" pour la vue actuelle

> Le moteur v2 doit être **au minimum aussi bon que la v1**, et idéalement meilleur (B).

2) Contrat d'entrée

Entrées "physiques"

Le moteur reçoit un `job_id` et récupère :

- les fichiers uploadés liés au job (`UnminifyFile`)
- leurs chemins (répertoire de storage du job)
- éventuellement un "mode" (par défaut `full`)

Types supportés v2

- `.js` , `.mjs` , `.cjs`
- `.css`
- `.html` , `.htm`
- `.json` (option : formatage)
- `.map` (option : extraction infos si présent)
- `.zip` (option : si tu l'avais prévu au CDCF, sinon v2.1)

Paramètres de job (optionnels)

Stockés soit :

- dans un champ `options_json` (si on l'ajoute plus tard)
- soit en dur v2.0 (full pipeline)

ETAPE NO 3

3) Contrat de sortie (100% compatible UI)

Dans le **storage_dir du job** (le même que v1), le moteur génère :

A) Progress

- `progress.json` **écrit en continu**
  - toujours lisible même en cours (écriture atomique)
  - format stable

Format JSON proposé

json

Copier le code

```
{
  "status": "queued|running|done|error",
  "percent": 0,
  "stage": "init|scan|beautify|analyze|report|package|finalize",
  "message": "string court lisible",
  "counters": {
    "files_total": 0,
    "files_done": 0,
    "js_done": 0,
    "css_done": 0,
    "html_done": 0,
    "errors": 0,
    "warnings": 0
  },
  "timings": {
    "started_at": "ISO8601",
    "updated_at": "ISO8601"
  }
}
```

> **Compat** : si ton front attend juste `percent` + `status`, ça marche. Sinon on adapte *sans casser* (on garde les clés historiques nécessaire).

B) Report HTML principal

- `report.html` (consommé tel quel par `report_view`)
  Contenu v2 :
- résumé global (nb fichiers, types, taille totale)
- tableau "par fichier" (lien vers artefacts)
- sections analysis JS (endpoints, patterns, strings, functions)
- erreurs/warnings

C) Artefacts par fichier

Pour chaque fichier `X` :

- `artifacts/beautified/X.beautified.js` (ou `.css` / `.html`)
- `artifacts/original/X.original.ext` (copie, si pas déjà)
- `artifacts/meta/X.meta.json` (stats : lignes, taille, hash, type, etc.)
- `artifacts/js_analysis/X.analysis.json` (si JS)

D) Package final

- `artifacts.zip` (ou `job_<id>_artifacts.zip`) accessible via la vue download existante

ETAPE NO 4

4) Pipeline v2 (stages)

Stage 0 — init

- créer l'arborescence :
  - `artifacts/`
  - `artifacts/beautified/`
  - `artifacts/meta/`
  - `artifacts/js_analysis/`
- écrire `progress.json` (status=running, stage=init, percent=1)

Stage 1 — scan

- lister fichiers du job
- détecter type + taille
- calculer hash (sha256) pour dédup interne
- update progress

Stage 2 — beautify (unminify)

- HTML : pretty print / normalisation (tolérant)
- CSS : formatage (au mieux) + règles safe
- JS : beautify (indentation + retours ligne) + fallback "heuristique" si parsing partiel

> Le but v2.0 : **lisible** + **stable**, pas "déobfuscation magique".

Stage 3 — analyze (reverse JS “safe”)

Pour JS, produire :

- endpoints / URLs / domaines (regex + heuristiques)
- détection patterns risqués : `eval`, `Function`, `setTimeout("...")`, wasm, crypto suspects
- extraction strings “intéressantes” (URLs, keys, tokens-like)
- top-level symbols : fonctions/classes/export si détectables
- stats : nombre de fonctions, longueur max ligne, etc.

Stage 4 — report

- générer `report.html` depuis un template (HTML propre)
- inclure liens vers les artefacts
- inclure warnings/errors

Stage 5 — package

- zipper `artifacts/` en un zip stable
- update progress

Stage 6 — finalize

- status=done, percent=100
- timestamp final

ETAPE NO 5

5) Exigences non-fonctionnelles

Robustesse

- **Jamais planter** sur 1 fichier : on marque "error" pour ce fichier et on continue.
- `progress.json` **écriture atomique** (write temp + rename).

Sécurité

- ne jamais exécuter le JS
- limiter taille max par fichier + taille totale job (anti zip bomb plus tard si zip supporté)

Performance

- traitement streaming quand possible
- pour énormes JS : mode "analyze light" si >X MB (on le mettra en v2.0 ou v2.1)

ETAPE NO 6

6) Impacts code (pour l'étape 2, pas maintenant)

- remplacer la partie Node de `tasks.py` par `engine.run_job(job_id)`
- ajouter un module `unminify/engine/` (nouveau)
- éventuellement ajuster `job_status_api` si ton front attend un format spécifique (on le gardera compatible)

1) NOUVEAU — unminify/engine_v2.py

PATCH DEJA APPLIQUES (1 – 9)

- 1) NOUVEAU — unminify/engine_v2.py
- 2) NOUVEAU — unminify/progress_compat.py
- 3) MODIF — unminify/tasks.py
- PATCH 2 — MODIF unminify/views.py
- PATCH 3 — MODIF unminify/engine_v2.py
- PATCH 4 (unique, utile, sans toucher le front)
- PATCH 5 (unique, 1 seul fichier) — Beautify JS beaucoup plus propre
- PATCH 6 (unique, 1 seul fichier) — ZIP final + manifest.json
- PATCH 7 (unique, 1 seul fichier) — Mode “gros fichiers” (safe + rapide)
- PATCH 8 (unique, 1 seul fichier) — Section “Erreurs & Warnings”
- PATCH 9 (unique, 1 seul fichier) — Ajouter summary.txt (humain)

PATCH DEJA APPLIQUES (10 - 20)

- PATCH 10 (unique, adapté) — Afficher message + counters
- PATCH 11 (unique) — Bouton “Download artifacts.zip” direct
- PATCH 12 (unique, 1 seul fichier) — Extraction “endpoints réseau”
- PATCH 13 (unique) — Support & analyse des .map
- PATCH 14 (unique) — Support ZIP upload (dézip sécurisé + anti zip-bomb)
- PATCH 15 (unique) — Score “Minification level” + détection auto (JS/CSS)
- PATCH 16 (unique) — Détecter sourceMappingURL dans les JS
- PATCH 17 (unique) — Détection “minified probable” + badge dans le report
- PATCH 18 (unique) — Top 20 fichiers les plus minifiés + liens “artefacts”
- PATCH 19 (unique) — Détecter signatures de librairies (React/Vue/jQuery/..)
- PATCH 20 (unique) — Sourcemaps : afficher “Top sources” + alerte

PATCH DEJA APPLIQUES (21 - 31)

- PATCH 21 (unique) — Détection “secrets/token-like” (heuristique)
- PATCH 22 (unique) — IOCs sécurité : domains/IPs/ws URLs + regroupement
- PATCH 23 (unique) — Enrichir summary.txt avec Top hosts/domains
- PATCH 24 (unique) — Fix sm_url + is_probably_minified manquant
- PATCH 25 (unique, correctif) — Fix sm_url + ajout is_probably_minified
- PATCH 26 — INSERTION 1 (calculer js_summary)
- PATCH 27 (mini-correctif unique) — 1 ligne à ajouter
- PATCH 28 (unique) — Ajouter top_minified (Top 10) dans manifest.json
- PATCH 29 (réel, simple, 100% clair)
- PATCH 30 (unique) — Nettoyer _load_all_js_analyses() : agrégation IOCs/ports
- PATCH 31 doublons eliminés

FONCTIONS DE BASES ATTENDUES

DEJA REALISE

🎯 Ce que FAIT déjà correctement `engine_v2.py`

Ta v2 est **solide sur le papier**. Elle sait :

✅ Lire des fichiers JS/CSS/HTML
✅ Détecter minification (heuristique OK)
✅ Beautifier JS/CSS/HTML **sans Node**
✅ Extraire :

- URLs
- domains
- XHR / fetch / axios / $.ajax
- strings intéressantes
- tokens "suspects"
- sourcemaps
- IOC basiques

✅ Générer :

- artifacts/
- report HTML
- progress.json

👉 **Le cœur moteur n'est PAS cassé.**

CE QUI MANQUE

1️⃣❌ Un mode d'entrée clair (SOURCE)

Problème

Le moteur suppose :

```python
job.files.all()
uf.abs_path()
uf.relpath
```

➡️ Donc **il ne fonctionne PAS seul.**

Il dépend d'un **wrapper Django (UnminifyJob / UnminifyFile).**

Ce qui manque

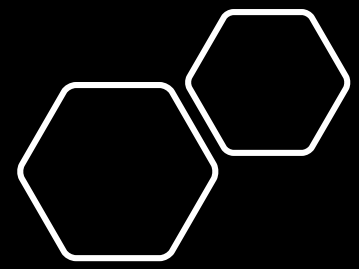
Un **mode standalone minimal**, par exemple :

- analyser un **répertoire**
- analyser une **URL (avec fetch Python)**
- analyser une **liste de fichiers**

Minimum vital à ajouter

Un **adaptateur d'entrée**, ex :

```python
run_engine_v2_from_dir("/tmp/site")
```



2️⃣❌ Un collecteur de fichiers JS réaliste

Problème

Le moteur **analyse ce qu'on lui donne**, mais :

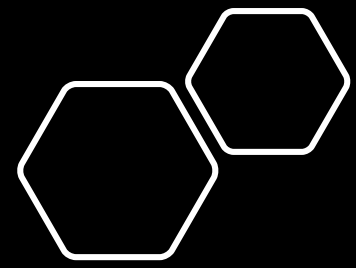
- il ne découvre pas les fichiers
- il ne crawl pas
- il ne suit pas `<script src=…>`

Ce qui manque

Un **pré-scan simple** :

- HTML → extraire `<script src>`
- CSS → extraire `@import`
- Optionnel : suivre 1 niveau de dépendances

👉 Sinon tu analyses **1 JS isolé**, pas un site.



3️⃣❌ Une gestion claire des erreurs JS

Problème

Le beautifier JS est **heuristique**, pas AST.

Il peut :

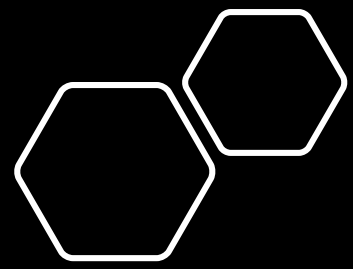
- casser du JS borderline
- exploser silencieusement
- produire du faux “joli”

Ce qui manque

Un **mode SAFE par défaut** :

- si JS > X Mo → pas de beautify
- si score minify < seuil → pas de beautify
- si beautify échoue → fallback = original

👉 Aujourd’hui, le moteur **essaie trop**, au lieu d’être conservateur.



4️⃣❌ Un résumé fonctionnel simple

Problème

Le report est **très riche**, mais pas lisible “au premier coup d’œil”.

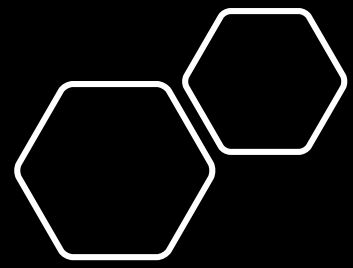
Ce qui manque (VRAIMENT)

Un **résumé de base**, en haut :

- combien de JS
- combien d’API détectées
- combien de domaines externes
- combien de sourcemaps
- combien de fichiers “à risque”

Sans jargon.

👉 C’est ce que l’humain attend **avant** de scroller 10 km.



5️⃣❌ Une séparation claire analyse vs rendu

Problème

`engine_v2.py` :

- analyse
- agrège
- génère HTML

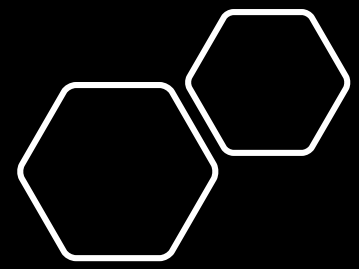
Tout est mélangé.

Ce qui manque

Même sans refactor lourd :

- `analysis/*.json` = vérité
- `report.html` = vue

👉 Aujourd'hui, si le HTML bug → tu perds la valeur.



6️⃣❌ Un mode CLI simple

Problème

Sans Django Admin + Celery, on est aveugle.

Ce qui manque

Un truc aussi simple que :

bash Copier le code

```bash
python engine_v2.py --dir ./site --out ./out
```

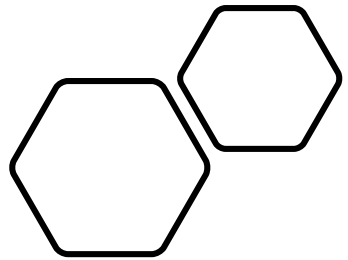
Ou même :

python Copier le code

```python
if __name__ == "__main__":
    run_engine_v2_standalone()
```

👉 Sinon ce n'est **pas un outil**, c'est une brique interne.

🧱 En résumé brutal (mais honnête)



Ce qui ne manque PAS

- ❌ Node
- ❌ NPM
- ❌ AST JS complexe
- ❌ Frameworks
- ❌ GraphQL
- ❌ React/Vue/etc.

Ce qui manque POUR DE VRAI

1. Entrée autonome (standalone)
2. Découverte des fichiers
3. Mode SAFE par défaut
4. Résumé humain lisible
5. Séparation analyse / rendu
6. CLI minimal