IDEO-Lab Technical Guide

Nuitka - Guide complet du compilateur Python

Installation, compilation, protection raisonnable du code, packaging standalone/onefile, modules importables, tests, CI/CD et cas reel Django addon.

Public vise
Developpeurs Python, Django, DevOps, editeurs de librairies, auteurs d'addons.

Objectif
Passer d'un premier script compile a un build industriel reproductible.

Focus
Compilation progressive, validation, protection du coeur metier, packaging propre.

Sommaire

Les cartes du guide HTML ont ete transformees en chapitres PDF lineaires. Chaque modale devient une section complete avec ses onglets deployes.

Section	Titre	Contenu
1.1	Vue d'ensemble Nuitka	Ce que Nuitka fait vraiment, ce qu'il ne fait pas, et dans quels cas il est pertinent.
1.2	Architecture & pipeline	Diagrammes du flux Python vers C/C++ puis binaire, avec modes de sortie.
2.1	Installation Windows/Linux	Prérequis compilateur C, venv, pip, cache, vérifications.
2.2	Quickstart script simple	Premier programme, compilation accélérée, standalone puis onefile.
3.1	Modes de compilation	Accelerated, standalone, onefile, module, package : comment choisir.
3.2	Options essentielles	Includes, excludes, plugins, data-files, report, output-dir, jobs.
4.1	Tests & validation	Comparer source vs binaire : sortie, exit code, imports, performances, packaging.
4.2	Cas réel : addon Django	Approche raisonnable pour compiler le cœur métier de ideolab_admin_tools.
5.1	Fichiers, templates, assets	Inclure JSON, YAML, templates, DLL, ressources package et imports dynamiques.
5.2	Protection du code	Obfuscation réelle ou protection raisonnable ? Limites, risques et architecture hybride.
6.1	Build industriel	Scripts build, Makefile, PowerShell, GitHub Actions, artefacts versionnés.
6.2	Dépannage	Imports manquants, antivirus, Python 3.13, MinGW/MSVC, chemins, DLL, performance.
7.1	Plan pas-à-pas	Approche progressive en 7 étapes pour valider que Nuitka est jouable.
7.2	Cheat-sheet	Commandes prêtes à copier pour Windows, Linux, standalone, onefile, modules.
7.3	Ressources	Liens officiels, documentation, pages de diagnostic et lectures recommandées.

Objectif du guide

Nuitka compile du Python vers du C/C++ puis produit un exécutable, un module compilé, un dossier standalone ou un fichier onefile. L'objectif pratique ici est de partir d'un script simple, puis d'aller vers un périmètre réaliste : un addon Python/Django dont on veut protéger le cœur sans transformer tout le projet en usine à gaz.

Python compiler · Standalone · Onefile · Django addon · Tests · CI/CD

Liens officiels utiles

Site	https://nuitka.net/
Manuel	User Manual
Tutorial	Setup and Build
PyPI	Nuitka on PyPI
GitHub	Nuitka GitHub
Common issues	Solutions to common issues

1.1 Vue d'ensemble Nuitka — Compilation, packaging, protection raisonnable

Nuitka en bref

Nuitka est un compilateur Python qui prend du code Python existant, l'analyse comme le ferait CPython, génère du code C/C++, puis délègue la production finale à un compilateur système comme **GCC**, **Clang**, **MSVC** ou **MinGW**.

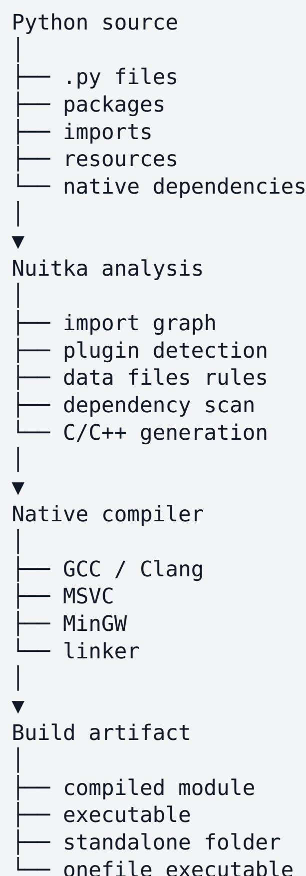
Son objectif n'est pas uniquement de produire un exécutable. Nuitka sert aussi à créer des livrables plus professionnels : modules compilés, dossiers autonomes, exécutables onefile, paquets plus difficiles à lire qu'un simple ensemble de fichiers `.py`.

> **Vision pratique :** Nuitka n'est pas un simple emballeur comme PyInstaller. Il transforme réellement le programme Python en une forme native intermédiaire, tout en cherchant à conserver une compatibilité très proche avec CPython.

Ce que Nuitka fait vraiment

- Analyse le graphe d'imports Python.
- Convertit les modules Python en code C/C++ généré.
- Compile ce code avec une toolchain native.
- Produit un artefact utilisable : module, exécutable, dossier standalone ou onefile.
- Inclut ou exclut des dépendances selon les options de build.
- Peut embarquer des fichiers data, DLL, extensions natives et packages tiers.

Carte mentale



```
Python source
│
├── .py files
├── packages
├── imports
├── resources
└── native dependencies
│
▼
Nuitka analysis
│
├── import graph
├── plugin detection
├── data files rules
├── dependency scan
└── C/C++ generation
│
▼
Native compiler
│
├── GCC / Clang
├── MSVC
├── MinGW
└── linker
│
▼
Build artifact
│
├── compiled module
├── executable
├── standalone folder
└── onefile executable
```

Positionnement rapide

Outil	Nature	Compilation réelle	Usage typique
CPython	Interpréteur	Non	Développement standard
PyInstaller	Packager	Non, principalement emballage	Livrer une app rapidement
Cython	Langage / transpiler	Oui	Extensions C, typage, performance ciblée
Nuitka	Compilateur Python	Oui	Packaging pro, protection source, compatibilité CPython

---

Pourquoi c'est intéressant pour un développeur Python senior ?

Dans un vrai projet Python, le problème n'est pas seulement d'exécuter le code. Le sujet devient rapidement : **comment livrer**, **comment protéger le cœur métier**, **comment éviter de distribuer toute l'arborescence source**, et **comment garantir un runtime stable**.

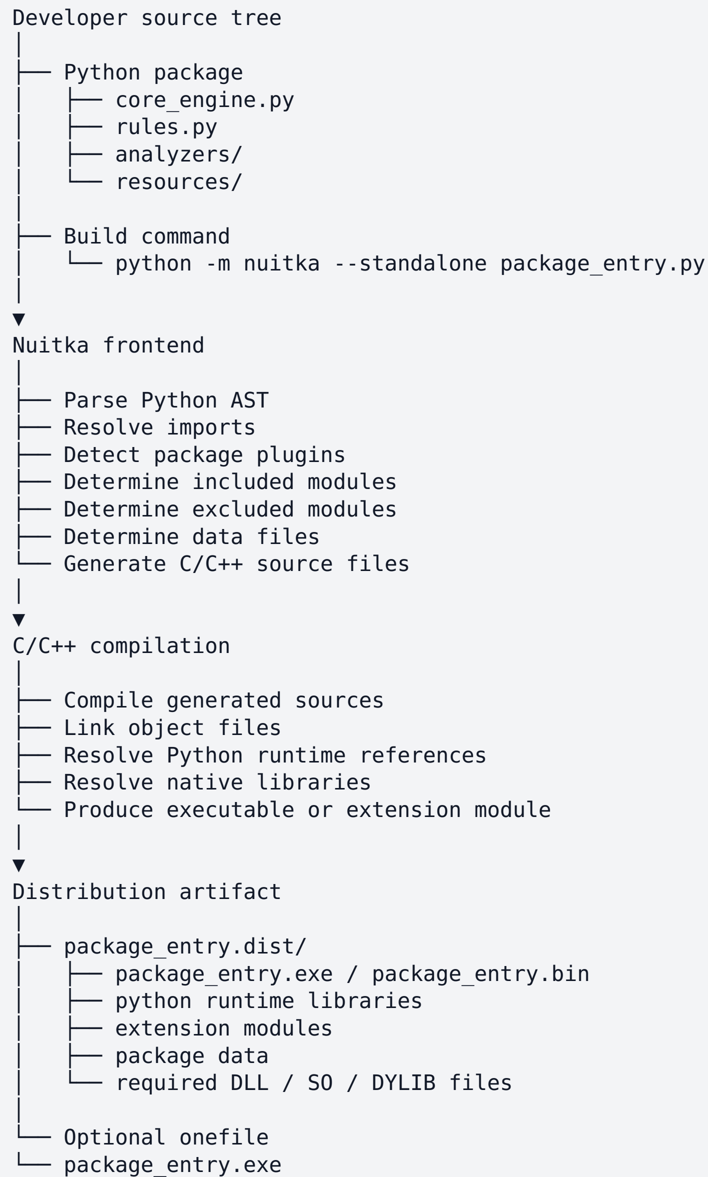
Nuitka permet une approche plus industrielle : on garde le projet Python lisible en interne, mais on compile les parties sensibles avant distribution.

Ce que Nuitka n'est pas

Idée reçue	Correction
Un outil magique anti reverse engineering	Non. Il augmente la difficulté, il ne rend pas l'analyse impossible.
Un remplacement de Python	Non. Le modèle d'exécution reste compatible avec l'écosystème Python.
Un accélérateur garanti	Non. Les gains dépendent fortement du type de code.
Un bouton unique pour compiler tout Django	Non. Il faut souvent compiler seulement un périmètre maîtrisé.

Pipeline complet de compilation Nuitka

Comprendre le pipeline est essentiel pour diagnostiquer les erreurs de build. Une erreur Nuitka peut venir du code Python, d'un import dynamique, d'une DLL manquante, du compilateur C/C++, ou d'un fichier data non inclus.



```
Developer source tree
│
├── Python package
│   ├── core_engine.py
│   ├── rules.py
│   ├── analyzers/
│   └── resources/
│
├── Build command
│   └── python -m nuitka --standalone package_entry.py
│
▼
Nuitka frontend
│
├── Parse Python AST
├── Resolve imports
├── Detect package plugins
├── Determine included modules
├── Determine excluded modules
├── Determine data files
└── Generate C/C++ source files
│
▼
C/C++ compilation
│
├── Compile generated sources
├── Link object files
├── Resolve Python runtime references
├── Resolve native libraries
└── Produce executable or extension module
│
▼
Distribution artifact
│
├── package_entry.dist/
│   ├── package_entry.exe / package_entry.bin
│   ├── python runtime libraries
│   ├── extension modules
│   ├── package data
│   └── required DLL / SO / DYLIB files
│
└── Optional onefile
└── package_entry.exe
```

Les 6 zones à surveiller

Zone	Risque	Contrôle
Imports	Module absent au runtime	`--include-module`, `--include-package`
Data files	Templates, JSON, YAML, assets absents	`--include-data-files`
DLL / SO	Librairie native non copiée	Tester sur machine propre
Plugins	Framework mal détecté	`--enable-plugin`
Compilateur	Toolchain absente ou incompatible	GCC, Clang, MSVC, MinGW
Runtime	Comportement différent du dev	Tests fonctionnels post-build

Diagnostic mental

```
Build fails
│
├── Syntax / Python error?
│   └── Run with CPython first
│
├── Compiler missing?
│   └── Install native compiler
│
├── Import missing?
│   └── Add include-module/package
│
├── Data missing?
│   └── Add include-data-files
│
├── Plugin needed?
│   └── Enable Nuitka plugin
│
└── Runtime-only crash?
└── Test standalone folder before onefile
```

> **Règle d'or :** toujours valider d'abord le mode `--standalone`. Le mode `--onefile` vient après, lorsque le dossier standalone fonctionne.

Les grands modes Nuitka

Nuitka n'a pas un seul usage. Il peut compiler un script, un module importable, un package complet, une application standalone ou un exécutable onefile. Le bon choix dépend du but : protection partielle, distribution client, outil CLI, intégration dans Django, ou livraison serveur.

Mode	Commande type	Résultat	Quand l'utiliser	Risque principal
Script simple	`python -m nuitka app.py`	Binaire local	Test minimal, preuve de concept	Dépendances non embarquées
Standalone	`python -m nuitka --standalone app.py`	Dossier `.dist`	Distribution fiable, test sérieux	Dossier volumineux
Onefile	`python -m nuitka --onefile app.py`	Un seul exécutable	Distribution simple à l'utilisateur final	Démarrage parfois plus lent
Module	`python -m nuitka --module engine.py`	Extension importable	Protéger un cœur métier importé par Python	Gestion imports / ABI
Package	`python -m nuitka --module mypackage`	Package compilé	Addon réutilisable, librairie propriétaire	Packaging Python plus délicat

Approche recommandée

```
Step 1: CPython validation
python -m pytest
python manage.py check

Step 2: minimal Nuitka build
python -m nuitka core_entry.py

Step 3: standalone build
python -m nuitka --standalone core_entry.py

Step 4: runtime validation
./core_entry.dist/core_entry

Step 5: optional onefile
python -m nuitka --standalone --onefile core_entry.py
```

Matrice de choix

Objectif	Mode conseillé
Tester Nuitka rapidement	Script simple
Livrer un outil CLI interne	Standalone
Livrer à un utilisateur non technique	Onefile après standalone
Protéger un moteur propriétaire	Module compilé
Compiler une app Django complète	À éviter au début

Piège fréquent : commencer directement par `--onefile`. C'est séduisant, mais plus difficile à déboguer. Le dossier standalone est beaucoup plus lisible.

Quand utiliser Nuitka ?

1. Distribuer un outil CLI

Idéal pour livrer une commande d'administration, un outil d'analyse, un scanner, un validateur, un convertisseur ou un agent local sans exposer directement tout le code source.

2. Protéger un cœur métier

Très pertinent pour compiler uniquement quelques modules sensibles : moteur de règles, scoring, parsing, heuristiques, analyseurs, génération de rapports, algorithmes propriétaires.

3. Stabiliser un livrable

Un dossier standalone peut contenir le runtime et les dépendances attendues, ce qui réduit les surprises liées aux environnements Python clients.

Cas adaptés à IDEO-Lab

Projet	Partie compilable	Intérêt
ideolab_admin_tools	Moteur d'introspection, scoring, détection admin	Protéger la valeur ajoutée de l'addon
Django Doctor	Analyseurs statiques, règles de diagnostic, graphes d'erreurs	Éviter de livrer les heuristiques brutes
SRDF	Parsing binlog, déduplication, compression, transport	Compiler les moteurs critiques, garder l'orchestration lisible
Migration Doctor	Classification d'erreurs, moteur de réparation	Protéger les algorithmes de diagnostic
Weglot-like i18n	Normalisation, déduplication, matching, rules engine	Protéger le cœur de traitement texte

Décision rapide

```
Is it business-critical?
│
├── No
│   └── Keep as normal Python
│
└── Yes
    │
    ├── Is it isolated?
    │   ├── Yes
    │   │   └── Compile with Nuitka
    │   │
    │   └── No
    │       └── Refactor into a small engine first
    │
    └── Does it need Django runtime?
        ├── No
        │   └── Better candidate
        └── Yes
            └── Compile carefully, test deeply
```

Meilleure stratégie : ne pas compiler tout le projet. Extraire un noyau stable, bien testé, avec peu de dépendances dynamiques, puis compiler ce noyau comme module importable.

Protection du code : vision réaliste

Nuitka apporte une protection nettement supérieure à la distribution de fichiers `.py` ou même `.pyc`, mais il ne faut pas vendre cela comme une sécurité absolue. Un attaquant très motivé peut analyser un binaire. Le vrai objectif est de rendre l'accès au cœur métier **plus coûteux**, **plus lent** et **moins immédiat**.

Niveaux de protection

Niveau	Distribution	Lisibilité	Protection
0	`.py`	Totale	Très faible
1	`.pyc`	Décompilable	Faible
2	Obfuscation Python	Réduite	Moyenne
3	Nuitka module	Faible	Bonne
4	Nuitka + architecture fermée	Très faible	Très bonne en pratique

Architecture de protection recommandée

```
public_package/
│
├── admin.py
├── apps.py
├── views.py
├── templates/
├── static/
│
├── public_api.py
│   └── imports compiled engine
│
└── private_core/
├── engine_compiled.pyd / .so
├── license_check.pyd / .so
└── rules_compiled.pyd / .so
```

> **Principe :** les parties Django visibles restent simples, tandis que la logique propriétaire est déplacée dans un noyau compilé.

Bonnes pratiques de protection

Bonne pratique	Pourquoi	Exemple
Compiler seulement le noyau	Moins de problèmes d'imports et de runtime	`rules_engine.py`, `analyzer_core.py`
Réduire les imports dynamiques	Nuitka analyse mieux les imports explicites	Éviter `importlib.import_module(name)` sans contrôle
Ajouter une API stable	Le reste du projet ne dépend pas des détails internes	`from private_core import analyze`
Tester sur machine propre	Détecte les dépendances oubliées	VM, conteneur, serveur vierge
Ne pas promettre l'inviolabilité	Un binaire peut toujours être étudié	Parler de protection raisonnable

---

Nuitka dans un projet Django : approche professionnelle

Compiler un site Django complet est possible dans certains contextes, mais ce n'est généralement pas le meilleur premier objectif. Django utilise beaucoup de conventions, d'imports dynamiques, de templates, de fichiers statiques, de settings, de migrations et de découverte automatique. L'approche robuste consiste à compiler une brique isolée.

Approche recommandée pour un addon Django

```
ideolab_admin_tools/
│
├── __init__.py
├── apps.py
├── admin.py
├── urls.py
├── views.py
├── templates/
├── static/
│
├── public_api.py
│
└── private_core/
├── __init__.py
├── admin_scanner.py
├── model_introspection.py
├── scoring.py
├── report_builder.py
└── license_rules.py
```

Ici, `admin.py`, `views.py` et les templates restent classiques. Le dossier `private_core` peut être progressivement compilé.

Ce qu'il faut éviter au début

À éviter	Raison
Compiler tout le projet Django	Surface trop grande, debug difficile
Compiler les migrations	Peu d'intérêt, risque inutile
Compiler les templates	Django les charge comme fichiers data
Compiler avant les tests	On ne sait plus si l'erreur vient du code ou du build
Compiler des modules instables	Chaque changement impose un nouveau cycle build/test

---

Pattern cible : façade Python + moteur compilé

```
Django Admin / Views / Commands
│
├── readable integration layer
│    ├── admin.py
│    ├── views.py
│    ├── management commands
│    └── templates
│
└── protected business core
├── compiled analyzer
├── compiled scoring
├── compiled rules
└── compiled report logic
```

Exemple de build module

```
python -m nuitka \
--module ideolab_admin_tools/private_core/scoring.py \
--output-dir=build/nuitka
```

Exemple de build standalone CLI

```
python -m nuitka \
--standalone \
--output-dir=build/nuitka \
tools/admin_scan_cli.py
```

Ce que Nuitka ne garantit pas

Important : un exécutable natif peut toujours être étudié par reverse engineering. Nuitka rend la récupération du code Python beaucoup moins directe, mais il ne transforme pas un algorithme en secret mathématiquement inviolable.

Point	Réalité	Action recommandée
Protection source	Meilleure que `.py` / `.pyc`, pas absolue	Compiler le noyau critique, garder une architecture fermée
Performance	Variable selon le type de code	Mesurer avant / après avec benchmarks simples
Imports dynamiques	Peuvent manquer dans le build	Utiliser `--include-module` ou `--include-package`
Fichiers data	Non inclus automatiquement dans tous les cas	Déclarer explicitement templates, JSON, YAML, assets
Django complet	Possible mais souvent complexe	Compiler une sous-brique plutôt que tout le site
Build cross-platform	Compiler Windows depuis Linux n'est pas le cas simple	Builder sur l'OS cible ou utiliser CI par plateforme
Antivirus	Certains onefile peuvent déclencher des alertes	Signer le binaire, préférer standalone en entreprise

Symptômes fréquents

- **ModuleNotFoundError :** import dynamique non détecté.
- **FileNotFoundError :** fichier data non inclus.
- **DLL load failed :** dépendance native manquante.
- **Crash seulement en onefile :** extraction temporaire ou chemin relatif incorrect.
- **Comportement différent :** dépendance au working directory ou à `__file__`.

Méthode de debug

```
1. Run with CPython
2. Run tests
3. Build without onefile
4. Run from .dist folder
5. Check missing imports
6. Check missing data files
7. Check native dependencies
8. Only then build onefile
```

Roadmap professionnelle pour adopter Nuitka

La meilleure façon de réussir Nuitka est de ne pas le traiter comme un simple bouton de compilation. Il faut avancer par paliers, avec un périmètre réduit, des tests reproductibles, puis une extension progressive.

Étape	Objectif	Action	Critère de réussite
Step 1	Valider l'environnement	Installer Nuitka + compiler un script hello world	Build OK + exécution OK
Step 2	Choisir un petit noyau	Identifier un module métier isolé	Module testable sans Django complet
Step 3	Créer une façade propre	Ajouter une API Python stable	Le reste du projet ignore les détails internes
Step 4	Compiler en module	Utiliser `--module`	Import OK depuis Python
Step 5	Compiler en standalone si besoin	Créer un outil CLI ou un binaire	Fonctionne sur machine propre
Step 6	Automatiser	Script build + dossier dist + rapport	Build reproductible

---

Checklist avant compilation

- Le module est stable.
- Le module a des tests unitaires.
- Les imports sont explicites.
- Les fichiers data sont identifiés.
- Les dépendances natives sont connues.
- Le module peut être testé hors interface web.
- Le build est documenté dans un script.

Checklist après compilation

- Le binaire ou module s'exécute.
- Les logs sont lisibles.
- Les erreurs sont capturées proprement.
- Les tests métier passent.
- Le livrable fonctionne hors environnement dev.
- La taille du package est acceptable.
- La procédure de rebuild est claire.

Liens utiles et documentation

Ces liens sont à conserver dans le guide pour permettre à l'utilisateur de creuser rapidement. Ils couvrent l'installation, les options, les problèmes fréquents et les cas avancés.

Ressource	URL	Utilité
Site officiel Nuitka	https://nuitka.net/	Point d'entrée officiel du projet.
User Manual	https://nuitka.net/user-documentation/user-manual.html	Options principales, modes de build, configuration.
PyPI Nuitka	https://pypi.org/project/Nuitka/	Installation pip, versions publiées, métadonnées package.
GitHub Nuitka	https://github.com/Nuitka/Nuitka	Sources, issues, discussions, releases.
Commercial / Nuitka Onefile / Support	https://nuitka.net/pages/commercial.html	Informations commerciales et fonctionnalités avancées.

Commandes de départ

```
python -m pip install -U nuitka ordered-set zstandard

python -m nuitka --version

python -m nuitka app.py

python -m nuitka --standalone app.py

python -m nuitka --standalone --onefile app.py
```

Mini-conclusion

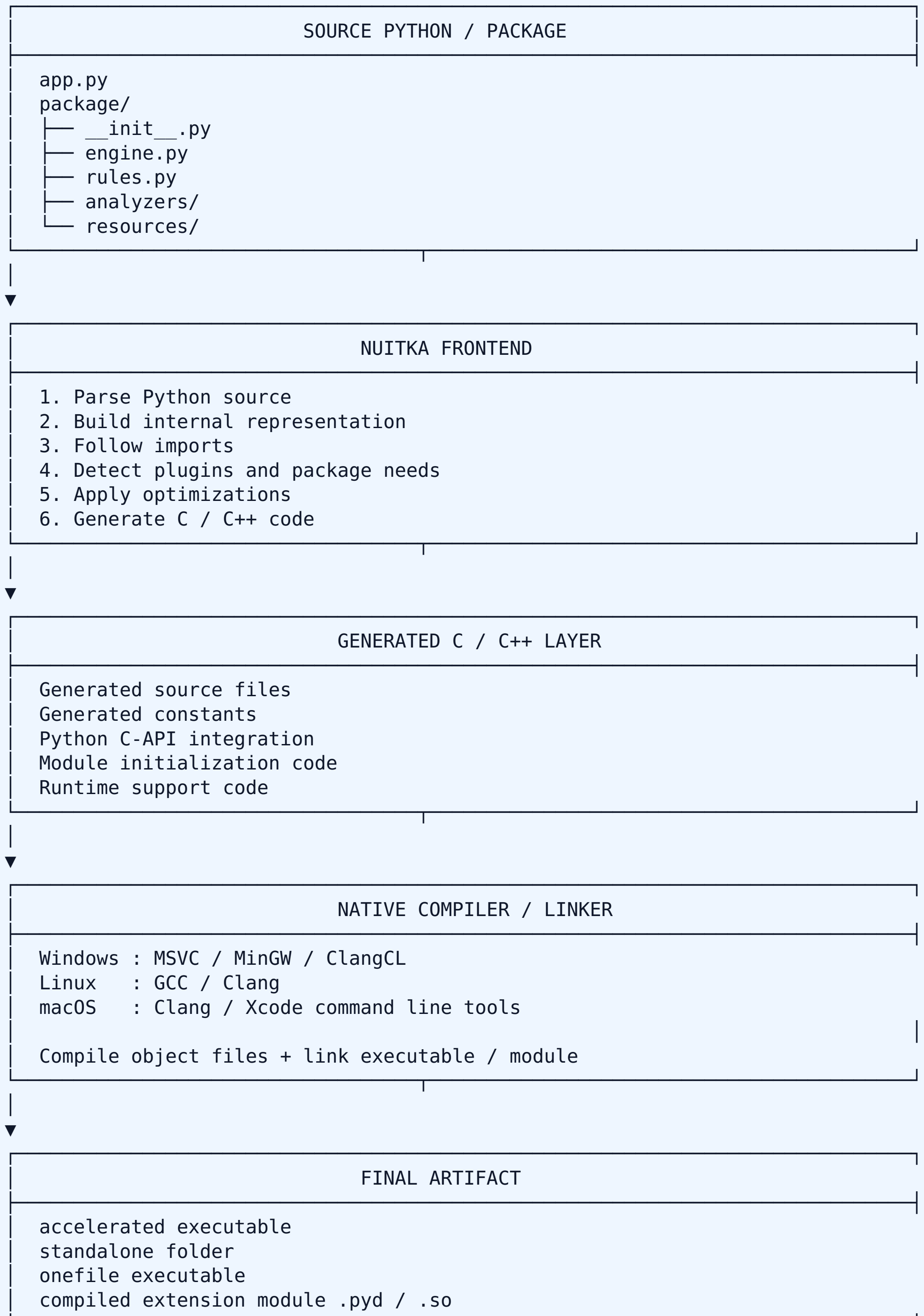
Nuitka est pertinent dès qu'un projet Python contient une valeur métier que l'on souhaite distribuer sans exposer directement les sources. La stratégie professionnelle consiste à compiler progressivement, en commençant par un petit périmètre très testé.

Pour IDEO-Lab : le meilleur premier terrain d'essai est un addon raisonnable, comme `ideolab_admin_tools`, avec compilation du cœur d'analyse plutôt que de toute la couche Django.

1.2 Architecture & pipeline Nuitka — du Python au binaire natif

Pipeline général de compilation

Nuitka fonctionne comme une chaîne de compilation en plusieurs couches. Il ne se contente pas de regrouper des fichiers Python : il analyse le programme, construit un graphe de dépendances, génère du code C/C++, puis utilise une toolchain native pour produire un artefact exécutable ou importable.



Lecture simple

Il faut voir Nuitka comme un **orchestrateur de build** : il connaît Python, il génère du C/C++, puis il pilote un compilateur natif. Les erreurs peuvent donc venir de plusieurs couches.

Couche	Responsabilité	Exemple d'erreur
Python	Code source, syntaxe, imports	`ModuleNotFoundError`
Nuitka	Analyse, plugins, génération C/C++	Import dynamique non détecté
Compilateur	Compilation native	MSVC, GCC ou Clang absent
Runtime	Exécution de l'artefact final	DLL / fichier data manquant

Point essentiel

Il y a deux compilations conceptuelles :
1. Nuitka compile Python vers une représentation native générée.
2. Le compilateur système compile cette représentation vers un binaire.

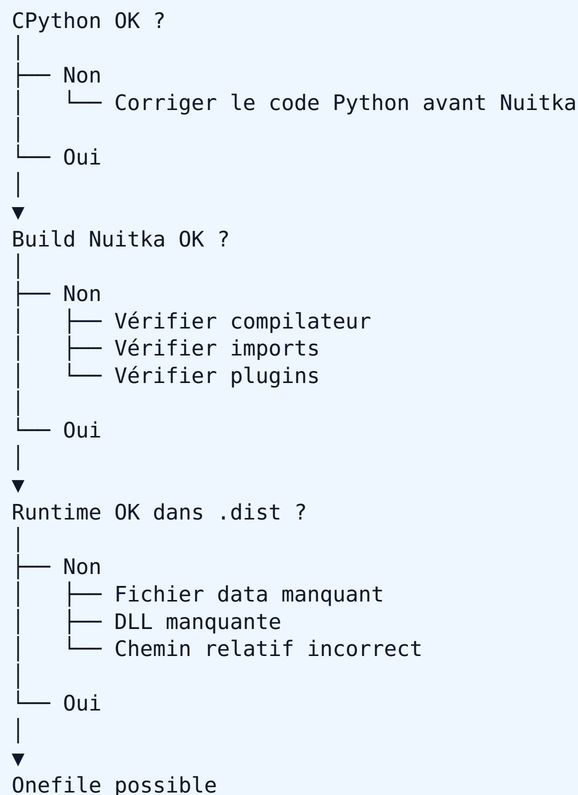
```
# Exemple minimal
python -m nuitka app.py

# Exemple plus réaliste
python -m nuitka --standalone app.py

# Exemple de livraison onefile
python -m nuitka --standalone --onefile app.py
```

Étapes internes du build Nuitka

Le build Nuitka peut être compris comme une suite d'étapes techniques. Cette lecture est très utile pour diagnostiquer les builds qui échouent, ou les exécutables qui fonctionnent sur la machine de développement mais pas ailleurs.

#	Étape	Ce qui se passe	Ce qu'il faut vérifier
1	**Entrée**	Nuitka reçoit un script, un module ou un package.	Le point d'entrée est clair et exécutable avec CPython.
2	**Parsing**	Le code Python est lu, parsé et transformé en représentation interne.	Aucune erreur de syntaxe, pas de code cassé masqué.
3	**Import graph**	Nuitka suit les imports directs et tente de résoudre les dépendances.	Les imports dynamiques sont explicitement inclus si nécessaire.
4	**Plugins**	Certains frameworks ou packages nécessitent des règles particulières.	Activer les plugins utiles si Nuitka ne les active pas automatiquement.
5	**Optimisation**	Nuitka applique ses optimisations et prépare le code généré.	Mesurer, ne pas supposer un gain de performance automatique.
6	**Codegen**	Génération de code C/C++ et des fichiers nécessaires au build.	Surveiller la taille, les logs et les warnings.
7	**Native compile**	Le compilateur système compile et linke les fichiers générés.	Toolchain installée, compatible avec Python utilisé.
8	**Distribution**	Nuitka prépare le dossier final ou l'exécutable final.	Présence des DLL, fichiers data, extensions natives, assets.
9	**Runtime test**	Le livrable est lancé comme le ferait un utilisateur final.	Tester hors venv, hors dossier source, sur machine propre.

Pipeline mental de validation



Messages à surveiller

- **Warning d'import :** souvent lié à un import dynamique ou optionnel.

- **Warning plugin :** indique qu'un package a besoin d'un traitement spécifique.
- **Erreur compilateur :** toolchain absente, mal configurée ou incompatible.
- **Erreur de link :** symbole ou librairie native introuvable.
- **Erreur runtime :** dépendance présente au build mais absente au lancement.

> **Bonne pratique :** conserver les logs de build dans un fichier. Sur un projet professionnel, un build non traçable est difficile à fiabiliser.

Artefacts produits par Nuitka

Le choix du mode de sortie détermine la forme du livrable, la facilité de debug, la taille finale, et le niveau de simplicité pour l'utilisateur final.

Mode	Commande type	Sortie	Usage recommandé	Avantage	Limite
Accelerated	`python -m nuitka app.py`	Exécutable lié à l'environnement local	Test rapide	Simple pour démarrer	Pas un vrai livrable autonome
Standalone	`python -m nuitka --standalone app.py`	Dossier `app.dist/`	Distribution contrôlée	Debug facile, fichiers visibles	Dossier volumineux
Onefile	`python -m nuitka --standalone --onefile app.py`	Un seul exécutable	Livraison utilisateur final	Très simple à distribuer	Debug plus difficile, extraction temporaire
Module	`python -m nuitka --module engine.py`	`.pyd` Windows ou `.so` Linux	Protéger un cœur métier importé	Très adapté aux addons	Packaging Python à soigner
Package module	`python -m nuitka --module mypackage`	Package compilé	Librairie propriétaire	Structure plus professionnelle	Plus complexe à distribuer

Anatomie d'un dossier standalone

```
app.dist/
│
├── app.exe / app.bin
├── python runtime libraries
├── extension modules
├── package dependencies
├── native libraries
├── data files
├── DLL / SO / DYLIB
└── optional resources
```

Le dossier standalone est le meilleur format pour comprendre ce que Nuitka embarque. C'est le format idéal pour les premiers tests sérieux.

Onefile : ce qui se passe au lancement

```
onefile executable
│
├── Start
├── Extract internal payload
├── Prepare temporary runtime folder
├── Launch embedded application
├── Execute Python-compiled program
└── Cleanup depending on options
```

Conséquence : un onefile peut démarrer moins vite qu'un standalone, car il doit préparer son environnement d'exécution.

Imports, graphe de dépendances et plugins

La compilation Python est fortement dépendante du graphe d'imports. Les imports explicites sont faciles à suivre. Les imports dynamiques, les plugins internes, les auto-discoveries et les chargements conditionnels demandent plus d'attention.

Graphe d'imports simple

```
app.py
│
├── import core.engine
│   ├── import core.rules
│   ├── import core.scoring
│   └── import core.formatters
│
├── import click
└── import pathlib

Nuitka peut suivre ce graphe facilement.
```

Import dynamique plus risqué

```
plugin_name = settings.SELECTED_PLUGIN

module = importlib.import_module(plugin_name)

module.run()

Nuitka ne peut pas toujours deviner
la valeur réelle de plugin_name.
```

Situation	Risque	Option / stratégie
Import explicite	Faible	Rien de spécial en général
Import via string	Moyen à fort	`--include-module=package.module`
Package complet nécessaire	Moyen	`--include-package=package_name`
Plugin framework	Variable	`--enable-plugin=plugin_name`
Import optionnel	Peut créer des warnings	Tester les chemins réellement utilisés

Situation	Risque	Option / stratégie
Django autodiscover	Fort si projet complet compilé	Préférer compiler un sous-module stable

Approche pro : réduire les imports dynamiques dans le cœur compilé. Une API claire, des imports explicites et des tests unitaires rendent Nuitka beaucoup plus prévisible.

Fichiers data : templates, JSON, YAML, assets, certificats

Nuitka compile du code, mais un projet réel contient aussi des fichiers qui ne sont pas du Python : templates, fichiers JSON, configurations YAML, images, dictionnaires, règles métier, certificats, fichiers SQL, modèles ML, etc.

```
Python code is not the whole application
│
├── code
│   ├── .py
│   └── packages
│
├── data
│   ├── templates/
│   ├── static/
│   ├── rules/*.json
│   ├── config/*.yaml
│   ├── certificates/*.pem
│   └── migrations/*.sql
│
└── native dependencies
├── DLL
├── SO
└── DYLIB
```

Exemples d'inclusion

```
# Inclure un fichier spécifique
python -m nuitka --standalone app.py \
--include-data-files=config/default.yaml=config/default.yaml

# Inclure un dossier de ressources
python -m nuitka --standalone app.py \
--include-data-dir=resources=resources

# Inclure des règles JSON
python -m nuitka --standalone app.py \
--include-data-files=rules/*.json=rules/
```

Table de contrôle

Type	Exemple	À vérifier
Template	`templates/report.html`	Inclus dans le build
Config	`config.yaml`	Chemin correct au runtime
Rules	`rules.json`	Version livrée correcte
Certificat	`cert.pem`	Présent et sécurisé
Static	`css/js/images`	Chemins compatibles onefile

Piège classique : le programme fonctionne depuis le code source, mais échoue après compilation car il lit un fichier par chemin relatif qui n'existe pas dans le livrable final.

Toolchain native : le deuxième moteur du build

Nuitka ne travaille pas seul. Il s'appuie sur un compilateur C/C++ installé sur la machine. Cette dépendance est normale : Nuitka génère du code natif, puis la toolchain système le compile.

OS	Compilateur typique	Installation / remarque	Symptôme si absent
Windows	MSVC, MinGW64, ClangCL	Visual Studio Build Tools ou MinGW selon environnement	Erreur indiquant compilateur introuvable
Linux	GCC ou Clang	`sudo apt install build-essential`	`gcc: command not found`
macOS	Clang	Xcode Command Line Tools	`xcrun` ou `clang` introuvable

Linux / Ubuntu

```
sudo apt update
sudo apt install -y python3 python3-venv python3-pip
sudo apt install -y build-essential ccache patchelf

python3 -m venv venv
source venv/bin/activate
python -m pip install -U pip nuitka ordered-set zstandard

python -m nuitka --version
```

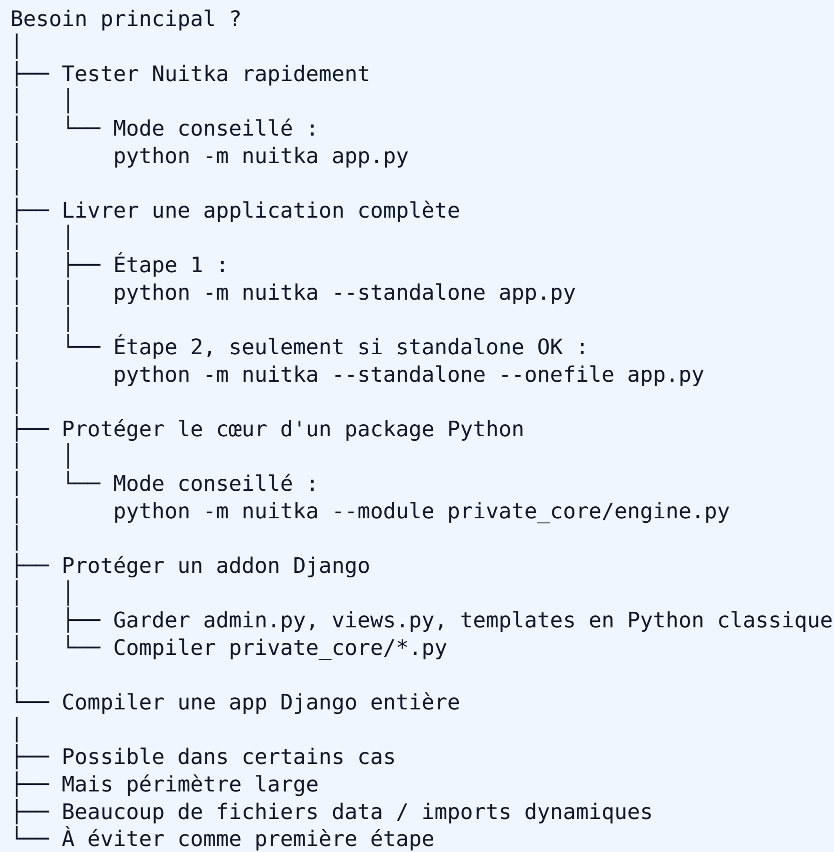
Windows

```
# 1. Installer Python 64-bit
# 2. Installer Visual Studio Build Tools
# 3. Ouvrir un terminal Developer ou PowerShell normal

python -m venv venv
venv\Scripts\activate

python -m pip install -U pip nuitka ordered-set zstandard

python -m nuitka --version
```

Pourquoi ccache peut aider ?

Sur Linux et certains environnements, `ccache` peut accélérer les recompilations en réutilisant des résultats de compilation lorsque les fichiers générés n'ont pas changé. C'est très utile quand on itère souvent sur un module compilé.

Arbre de décision : quel mode choisir ?



```
Besoin principal ?
│
├── Tester Nuitka rapidement
│   │
│   └── Mode conseillé :
│       python -m nuitka app.py
│
├── Livrer une application complète
│   │
│   ├── Étape 1 :
│   │   python -m nuitka --standalone app.py
│   │
│   └── Étape 2, seulement si standalone OK :
│       python -m nuitka --standalone --onefile app.py
│
├── Protéger le cœur d'un package Python
│   │
│   └── Mode conseillé :
│       python -m nuitka --module private_core/engine.py
│
├── Protéger un addon Django
│   │
│   ├── Garder admin.py, views.py, templates en Python classique
│   └── Compiler private_core/*.py
│
└── Compiler une app Django entière
    │
    ├── Possible dans certains cas
    ├── Mais périmètre large
    ├── Beaucoup de fichiers data / imports dynamiques
    └── À éviter comme première étape
```

Matrice de décision

Contexte	Meilleur choix	Pourquoi
Petit script CLI	Standalone	Facile à tester et livrer
Outil client final	Onefile après standalone	Distribution simple
Librairie propriétaire	Module	Protection ciblée
Addon Django	Core compilé seulement	Évite de casser la mécanique Django
Service serveur	Standalone ou module	Dépend du mode de déploiement

Signaux d'alerte

- Le module utilise massivement `importlib`.
- Le module lit beaucoup de fichiers par chemins relatifs.
- Le module dépend fortement des settings Django.
- Le module n'a aucun test automatisé.
- Le module change tous les jours.
- Le build doit fonctionner sur plusieurs OS sans CI.

> **Conclusion :** plus le périmètre est petit, stable et explicite, plus Nuitka est simple à réussir.

Debug du pipeline : lire les pannes par couche

Pour éviter de perdre du temps, il faut classer l'erreur selon la couche concernée : Python, Nuitka, compilateur, packaging ou runtime.

Symptôme	Couche probable	Cause fréquente	Action
Le script échoue avant Nuitka	Python	Erreur applicative réelle	Corriger sous CPython avant compilation
Nuitka ne trouve pas un module	Analyse imports	Import dynamique ou package optionnel	Ajouter `--include-module` ou `--include-package`
Erreur MSVC / GCC / Clang	Compilation native	Toolchain absente ou mal configurée	Installer le compilateur et relancer
L'exécutable démarre puis crash	Runtime	DLL, data file ou chemin manquant	Tester dans `.dist`, inspecter les fichiers
Standalone OK mais onefile KO	Packaging onefile	Chemin temporaire, extraction, working directory	Corriger les accès fichiers, éviter chemins relatifs fragiles

Procédure de debug recommandée

```
# 1. Valider Python pur
python app.py

# 2. Lancer les tests
python -m pytest

# 3. Build simple
python -m nuitka app.py

# 4. Build standalone
python -m nuitka --standalone app.py

# 5. Exécuter depuis le dossier dist
cd app.dist
./app

# 6. Seulement ensuite : onefile
python -m nuitka --standalone --onefile app.py
```

Rapport de build conseillé

```
build_report/
│
├── command.txt
├── nuitka_version.txt
├── python_version.txt
├── platform.txt
├── build.log
├── runtime_test.log
├── included_modules.txt
├── included_data_files.txt
└── known_issues.md
```

Approche professionnelle : chaque build important doit être reproductible. Il faut garder la commande, la version Python, la version Nuitka et les logs.

2.1 Installation Windows / Linux — environnement propre, compilateur C, venv, vérifications

Objectif de cette étape

Avant de compiler un vrai projet avec Nuitka, il faut valider une base saine : **Python correct**, **environnement virtuel isolé**, **pip à jour**, **Nuitka installé**, **compilateur C/C++ détecté**, et **premier build fonctionnel**.

Chaîne d'installation complète

```
Machine
│
├── Python installé
│   ├── python --version
│   └── python -m pip --version
│
├── Environnement virtuel
│   ├── .venv/
│   ├── pip isolé
│   └── packages projet isolés
│
├── Nuitka
│   ├── nuitka
│   ├── ordered-set
│   └── zstandard
│
├── Compilateur C/C++
│   ├── Windows : MSVC / Build Tools
│   ├── Linux   : gcc / g++ / make
│   └── macOS   : clang
│
└── Validation
├── python hello.py
├── python -m nuitka hello.py
└── executable généré OK
```

Pourquoi cette étape est critique ?

Élément	Pourquoi c'est important	Symptôme si absent
venv	Isole les dépendances du projet	Versions incohérentes, conflits pip
pip récent	Installe correctement les roues modernes	Builds longs ou erreurs d'installation
Nuitka	Compilateur Python vers C/C++	`No module named nuitka`
ordered-set	Optimise certaines étapes Nuitka	Build moins efficace
zstandard	Compression utile pour certains modes	Onefile / compression moins confortable
Compilateur C	Produit le binaire final	Erreur toolchain / compiler not found

Règle IDEO-Lab : on ne compile jamais un vrai addon avant d'avoir validé un build minimal `hello.py` sur la même machine.

Ordre recommandé

```
1. Vérifier Python
|
▼
2. Créer .venv
|
▼
3. Activer .venv
|
▼
4. Mettre pip / setuptools / wheel à jour
|
▼
5. Installer Nuitka + dépendances recommandées
|
▼
6. Installer / vérifier le compilateur C/C++
|
▼
7. Compiler hello.py
|
▼
8. Lancer le binaire généré
|
▼
9. Passer à un vrai module projet
```

Créer une base Python propre avec venv

Nuitka doit être installé dans un environnement Python clair. Pour éviter les conflits, on utilise un `.venv` dédié au projet ou au test.

Commandes universelles

```
# Créer un environnement virtuel local
python -m venv .venv

# Windows CMD
.venv\Scripts\activate.bat

# Windows PowerShell
.venv\Scripts\Activate.ps1

# Linux / macOS
source .venv/bin/activate

# Vérifier que le venv est actif
python -c "import sys; print(sys.executable)"
```

Contrôle visuel : après activation, le terminal affiche souvent `(.venv)` au début de la ligne de commande.

Installer Nuitka proprement

```
# Mettre à jour les outils de packaging Python
python -m pip install --upgrade pip setuptools wheel

# Installer Nuitka et dépendances recommandées
python -m pip install -U nuitka ordered-set zstandard

# Vérifier
python -m nuitka --version
python -m pip show nuitka
```

Commande	But
`pip`	Installer les packages Python
`setuptools`	Support packaging Python classique
`wheel`	Installer des distributions binaires
`nuitka`	Compiler Python vers C/C++ puis binaire
`ordered-set`	Accélérer / améliorer certains traitements internes
`zstandard`	Compression moderne, utile selon les builds

Arborescence recommandée

```
nuitka_lab/
│
├── .venv/
├── hello.py
├── requirements-build.txt
├── build/
│   ├── logs/
│   └── dist/
└── README_BUILD.md
```

À ne pas faire

- Installer Nuitka globalement sans savoir quel Python est utilisé.
- Mélanger plusieurs versions de Python dans le même terminal.
- Compiler depuis un dossier contenant des caractères bizarres ou chemins très longs.
- Compiler un gros projet sans test minimal préalable.
- Oublier de vérifier `sys.executable`.

Installation Windows — Python, MSVC, venv, Nuitka

Sur Windows, l'étape la plus importante est le compilateur C/C++. Pour une approche stable et professionnelle, privilégier **Microsoft Visual Studio Build Tools** avec le composant **Desktop development with C++**.

Plan Windows recommandé

```
Windows
│
├── Installer Python 64-bit
│   ├── Add Python to PATH
│   └── Vérifier python --version
│
├── Installer Visual Studio Build Tools
│   └── Desktop development with C++
│
├── Créer .venv
│   └── python -m venv .venv
│
├── Installer Nuitka
│   └── pip install -U nuitka ordered-set zstandard
│
└── Compiler hello.py
└── python -m nuitka hello.py
```

Commandes Windows

```
# Vérifier Python
python --version
python -c "import sys; print(sys.version); print(sys.executable)"

# Créer et activer le venv
python -m venv .venv

# CMD
.venv\Scripts\activate.bat

# PowerShell
.venv\Scripts\Activate.ps1

# Mettre pip à jour
python -m pip install --upgrade pip setuptools wheel

# Installer Nuitka
python -m pip install -U nuitka ordered-set zstandard

# Vérifier Nuitka
python -m nuitka --version
```

Visual Studio Build Tools

Élément	À sélectionner	Pourquoi
Workload	**Desktop development with C++**	Installe la toolchain C/C++ nécessaire
MSVC	Version récente compatible	Compile le code C/C++ généré
Windows SDK	SDK proposé par Visual Studio Installer	Nécessaire au linkage Windows
CMake tools	Optionnel selon projets	Utile pour certains packages natifs

Vérifications Windows

```
python --version
python -m pip --version
python -m nuitka --version

# Vérifier l'architecture Python
python -c "import platform; print(platform.architecture())"

# Vérifier où pointe Python
where python
where pip
```

Pièges Windows fréquents

Problème	Cause	Solution
PowerShell bloque l'activation	Execution policy restrictive	`Set-ExecutionPolicy RemoteSigned -Scope CurrentUser`
Mauvais Python utilisé	PATH ambigu	`where python`
Compilateur absent	Build Tools non installés	Installer Desktop development with C++
Chemins trop longs	Projet profondément imbriqué	Utiliser un chemin court comme `C:\dev\nuitka_lab`
Antivirus bloque le binaire	Onefile ou exe inconnu	Tester standalone, signer si livraison pro

> **Conseil de démarrage :** pour un premier test confortable sous Windows, partir sur Python 3.11 ou 3.12 64-bit + MSVC Build Tools est souvent plus simple que de démarrer avec une combinaison trop récente ou exotique.

Installation Linux / Ubuntu / Debian

Sur Linux, l'installation est généralement plus directe : Python, venv, pip, `build-essential`, puis Nuitka. Pour les builds standalone, `patchelf` est souvent utile.

Installation système

```
sudo apt update

sudo apt install -y \
python3 \
python3-venv \
python3-pip \
build-essential \
ccache \
patchelf
```

`build-essential` installe notamment `gcc`, `g++` et `make`. `patchelf` aide sur certains builds standalone Linux. `ccache` peut accélérer les recompilations.

Créer le venv et installer Nuitka

```
mkdir -p ~/nuitka_lab
cd ~/nuitka_lab

python3 -m venv .venv
source .venv/bin/activate

python -m pip install --upgrade pip setuptools wheel
python -m pip install -U nuitka ordered-set zstandard

python -m nuitka --version
```

Contrôles Linux

Contrôle	Commande	Résultat attendu
Python	`python3 --version`	Version Python affichée
Venv actif	`which python`	Chemin vers `.venv/bin/python`
pip actif	`python -m pip --version`	pip dans le venv
GCC	`gcc --version`	Version GCC affichée
G++	`g++ --version`	Version G++ affichée
Patchelf	`patchelf --version`	Version patchelf affichée
Nuitka	`python -m nuitka --version`	Version + informations environnement

Ubuntu 24 / serveur propre

```
sudo apt update
sudo apt install -y python3 python3-venv python3-pip
sudo apt install -y build-essential ccache patchelf

python3 -m venv .venv
source .venv/bin/activate

python -m pip install -U pip setuptools wheel
python -m pip install -U nuitka ordered-set zstandard
```

Pièges Linux fréquents

- `python` absent mais `python3` disponible.
- `venv` non installé : installer `python3-venv`.
- `gcc` absent : installer `build-essential`.
- `patchelf` absent : certains standalone peuvent échouer ou être incomplets.
- Build lancé hors venv : Nuitka ne voit pas les bons packages.

Cache, accélération et confort de build

Nuitka peut générer beaucoup de fichiers C/C++ puis les compiler. Sur des projets moyens ou grands, les temps de build peuvent devenir importants. Il faut donc mettre en place un minimum de confort : cache, dossier de build clair, logs et commandes reproductibles.

ccache sous Linux

```
sudo apt install -y ccache

ccache --version
ccache --show-stats

# Optionnel : voir le chemin
which ccache
```

`ccache` mémorise certains résultats de compilation. Si les fichiers générés n'ont pas changé, les recompilations peuvent être accélérées.

Dossier de build propre

```
build/
│
├── nuitka/
│   ├── hello.build/
│   ├── hello.dist/
│   └── hello.onefile-build/
│
├── logs/
│   ├── build_hello.log
│   └── runtime_hello.log
│
└── reports/
├── environment.txt
└── command_used.txt
```

Options utiles à connaître

Option	Usage	Remarque
`--output-dir=build/nuitka`	Centraliser les sorties de build	Évite de polluer la racine du projet
`--remove-output`	Nettoyer certains outputs précédents	Utile pour repartir propre
`--show-progress`	Afficher la progression	Pratique sur gros build
`--show-memory`	Afficher l'usage mémoire	Utile sur serveur limité
`--standalone`	Créer un dossier autonome	Format de debug recommandé
`--onefile`	Créer un exécutable unique	À utiliser après validation standalone

Conseil pro : dès qu'un build devient important, ne pas taper les commandes à la main. Créer un script `build_nuitka.py` ou `build_nuitka.sh` qui écrit un rapport de build.

Vérifications de l'environnement

Cette section sert de diagnostic rapide. Avant de signaler un bug Nuitka ou de compiler un vrai module, ces commandes doivent être exécutées et leurs résultats doivent être cohérents.

Checks Python / pip / Nuitka

```
python -c "import sys; print(sys.version); print(sys.executable)"
python -m pip --version
python -m pip list

python -m nuitka --version
python -m nuitka --help
```

Checks système

```
# Linux
which python
which gcc
gcc --version
g++ --version
patchelf --version

# Windows
where python
where pip
python --version
python -m nuitka --version
```

Tableau de validation

Test	Commande	OK si...	Correction si KO
Python visible	`python --version`	Une version s'affiche	Installer Python ou corriger PATH
Venv actif	`python -c "import sys; print(sys.executable)"`	Chemin contient `.venv`	Réactiver le venv
pip fonctionne	`python -m pip --version`	pip répond dans le venv	Réinstaller / mettre à jour pip
Nuitka installé	`python -m nuitka --version`	Version Nuitka affichée	`python -m pip install -U nuitka`
Compilateur présent	`gcc --version` ou MSVC	Version compilateur visible	Installer Build Tools ou build-essential
Build minimal OK	`python -m nuitka hello.py`	Un exécutable est produit	Lire l'erreur par couche

> **Important :** toujours utiliser `python -m nuitka` plutôt que `nuitka` au début. Cela garantit que Nuitka est lancé avec le Python du venv actif.

Premier build de validation : hello.py

Le premier build ne doit pas être un projet Django, un addon complexe ou un outil avec dépendances. On commence avec un fichier minimal, pour vérifier toute la chaîne : Python → Nuitka → C/C++ → binaire.

Créer le fichier de test

```
# Linux / macOS
cat > hello.py <<'PY'
import sys
import platform

print("hello from nuitka")
print("python:", sys.version)
print("executable:", sys.executable)
print("platform:", platform.platform())
PY
```

```
# Windows PowerShell
@'
import sys
import platform

print("hello from nuitka")
print("python:", sys.version)
print("executable:", sys.executable)
print("platform:", platform.platform())
'@ | Set-Content -Encoding UTF8 hello.py
```

Exécuter en Python pur

```
python hello.py
```

Cette étape valide que le script fonctionne avant toute compilation. Si cette commande échoue, le problème n'est pas Nuitka.

Compiler en mode simple

```
python -m nuitka hello.py
```

Build standalone puis onefile

Standalone recommandé

```
python -m nuitka \
--standalone \
--output-dir=build/nuitka \
hello.py

# Linux
./build/nuitka/hello.dist/hello

# Windows
build\nuitka\hello.dist\hello.exe
```

Onefile après validation

```
python -m nuitka \
--standalone \
--onefile \
--output-dir=build/nuitka \
hello.py

# Linux
./build/nuitka/hello.bin

# Windows
build\nuitka\hello.exe
```

Validation minimale réussie : le script fonctionne avec CPython, Nuitka compile sans erreur, et le binaire produit affiche le même résultat.

Erreurs fréquentes d'installation et corrections

Erreur / symptôme	Cause probable	Correction	Niveau
`No module named nuitka`	Nuitka non installé dans ce Python	`python -m pip install -U nuitka`	Simple
`pip not found`	pip absent ou PATH incorrect	`python -m ensurepip --upgrade`	Simple
PowerShell refuse `Activate.ps1`	Execution policy Windows	`Set-ExecutionPolicy RemoteSigned -Scope CurrentUser`	Windows
Compilateur C introuvable	MSVC / GCC absent	Installer Build Tools ou `build-essential`	Système
`gcc: command not found`	Toolchain Linux absente	`sudo apt install build-essential`	Linux
Build très lent	Pas de cache ou projet volumineux	Installer `ccache`, limiter le périmètre	Confort
Executable bloqué par antivirus	Binaire inconnu, onefile compressé	Tester standalone, signer le binaire si besoin	Windows
Fonctionne en source, pas en binaire	Data file ou DLL manquante	Ajouter include-data, tester dans `.dist`	Runtime

Diagnostic rapide

```
Erreur installation ?
│
├── Python introuvable
│   └── PATH / installation Python
│
├── Nuitka introuvable
│   └── pip install dans le bon venv
│
├── Compilateur introuvable
│   ├── Windows : Build Tools
│   └── Linux   : build-essential
│
├── Build échoue
│   └── Lire logs Nuitka
│
└── Runtime échoue
├── Tester standalone
├── Vérifier data files
└── Vérifier DLL / SO
```

Commande de rapport environnement

```
python - <<'PY'
import sys
import platform
import subprocess

print("Python:", sys.version)
print("Executable:", sys.executable)
print("Platform:", platform.platform())
print("Machine:", platform.machine())

try:
import nuitka
print("Nuitka import: OK")
except Exception as exc:
print("Nuitka import: KO", exc)

subprocess.run([sys.executable, "-m", "pip", "--version"])
PY
```

Setup professionnel pour projets IDEO-Lab

Pour un vrai projet comme `ideolab_admin_tools`, `Django Doctor`, `SRDF` ou `Migration Doctor`, il faut éviter les builds improvisés. L'installation doit produire un environnement reproductible.

Fichier requirements-build.txt

```
# requirements-build.txt
nuitka
ordered-set
zstandard
wheel
setuptools
pytest
```

Installation reproductible

```
python -m venv .venv
source .venv/bin/activate

python -m pip install --upgrade pip setuptools wheel
python -m pip install -r requirements-build.txt

python -m nuitka --version
```

Script build recommandé

```
scripts/
│
├── build_nuitka_module.py
├── build_nuitka_cli.py
├── check_nuitka_env.py
└── clean_nuitka_build.py

build/
│
├── nuitka/
├── logs/
└── reports/
```

Bon niveau professionnel : un développeur doit pouvoir cloner le projet, créer le venv, installer `requirements-build.txt`, lancer le script de build, et obtenir le même artefact.

Checklist finale avant de passer au vrai projet

#	Contrôle	Statut attendu
1	Python installé et identifié	`python -c "import sys; print(sys.executable)"` OK
2	Venv actif	Le chemin pointe vers `.venv`
3	pip à jour	`pip`, `setuptools`, `wheel` OK
4	Nuitka installé	`python -m nuitka --version` OK
5	Compilateur C/C++ disponible	MSVC ou GCC détecté
6	hello.py compile	Build simple OK
7	Standalone compile	Dossier `.dist` fonctionnel
8	Logs conservés	Commande et environnement documentés

Prochaine étape logique : choisir un petit module réel, stable, avec peu d'imports dynamiques, et le compiler en mode `--module` ou en mini CLI standalone.

2.2 Quickstart Nuitka — premier script, accelerated, standalone, onefile

Objectif du quickstart

Ce premier exercice sert à valider toute la chaîne Nuitka sur un script volontairement simple : exécution Python classique, compilation rapide, build standalone, puis build onefile. Le but n'est pas encore de compiler un projet Django ou un addon complexe, mais de vérifier que **l'environnement**, **le compilateur C/C++**, **Nuitka** et **le runtime généré** fonctionnent correctement.

Pipeline de ce quickstart

```
hello_nuitka.py
│
├── 1. Exécution CPython
│      python hello_nuitka.py
│
├── 2. Build accelerated
│      python -m nuitka hello_nuitka.py
│
├── 3. Build standalone
│      python -m nuitka --standalone hello_nuitka.py
│
├── 4. Build onefile
│      python -m nuitka --standalone --onefile hello_nuitka.py
│
└── 5. Comparaison
sortie source == sortie binaire
exit code == 0
artefacts présents
```

Ce que l'on valide

Contrôle	Pourquoi	Résultat attendu
Python source	Vérifier que le script est sain	Sortie texte OK
Nuitka frontend	Vérifier parsing + génération	Build sans erreur
Compilateur C	Vérifier toolchain native	Binaire produit
Standalone	Vérifier distribution autonome	Dossier `.dist` fonctionnel
Onefile	Vérifier livraison fichier unique	Exécutable unique fonctionnel

Règle importante : on ne passe au mode `onefile` qu'après avoir validé le mode `standalone`. Le dossier standalone est beaucoup plus simple à inspecter en cas de problème.

Organisation recommandée du laboratoire

```
nuitka_quickstart/
│
├── .venv/
├── hello_nuitka.py
├── build/
│   ├── nuitka/
│   ├── logs/
│   └── reports/
└── README_QUICKSTART.md
```

Créer le script `hello_nuitka.py`

Le script ci-dessous est volontairement simple, mais pas totalement vide. Il utilise quelques modules standards, une fonction métier minimale, un affichage système, un calcul déterministe et un code retour propre. Cela permet de vérifier que le binaire reproduit le comportement du script source.

Code Python complet

```
from __future__ import annotations

                                import platform
                                import sys
                                from datetime import datetime
                                from pathlib import Path


                                def compute_score(values: list[int]) -> int:
                                """
                                Small deterministic function used to verify that the compiled binary
                                returns the same business result as the Python source version.
                                """
                                return sum(value * 3 for value in values) + 17


                                def describe_runtime() -> dict[str, str]:
                                """
                                Return runtime information that helps compare CPython execution
                                and Nuitka compiled execution.
                                """
                                return {
                                "python": sys.version.split()[0],
                                "platform": f"{platform.system()} {platform.machine()}",
                                "executable": str(Path(sys.executable).name),
                                "cwd": str(Path.cwd()),
                                }


                                def main() -> int:
                                runtime = describe_runtime()
                                score = compute_score([1, 2, 3, 4])

                                print("Nuitka smoke test")
                                print(f"Python: {runtime['python']}")
                                print(f"Platform: {runtime['platform']}")
                                print(f"Executable: {runtime['executable']}")
                                print(f"Working directory: {runtime['cwd']}")
                                print(f"Time: {datetime.now().isoformat(timespec='seconds')}")
                                print(f"Score: {score}")

                                if score != 47:
                                print("Unexpected score")
                                return 2

                                return 0


                                if __name__ == "__main__":
                                raise SystemExit(main())
```

Créer le fichier rapidement

Linux / macOS

```
cat > hello_nuitka.py <<'PY'
                                from __future__ import annotations

                                import platform
                                import sys
                                from datetime import datetime
                                from pathlib import Path


                                def compute_score(values: list[int]) -> int:
                                return sum(value * 3 for value in values) + 17


                                def describe_runtime() -> dict[str, str]:
                                return {
                                "python": sys.version.split()[0],
                                "platform": f"{platform.system()} {platform.machine()}",
                                "executable": str(Path(sys.executable).name),
                                "cwd": str(Path.cwd()),
                                }


                                def main() -> int:
                                runtime = describe_runtime()
                                score = compute_score([1, 2, 3, 4])

                                print("Nuitka smoke test")
                                print(f"Python: {runtime['python']}")
                                print(f"Platform: {runtime['platform']}")
                                print(f"Executable: {runtime['executable']}")
                                print(f"Working directory: {runtime['cwd']}")
                                print(f"Time: {datetime.now().isoformat(timespec='seconds')}")
                                print(f"Score: {score}")

                                if score != 47:
                                print("Unexpected score")
                                return 2

                                return 0


                                if __name__ == "__main__":
                                raise SystemExit(main())
                                PY
```

Windows PowerShell

```
@'
                                from __future__ import annotations

                                import platform
                                import sys
                                from datetime import datetime
                                from pathlib import Path


                                def compute_score(values: list[int]) -> int:
                                return sum(value * 3 for value in values) + 17


                                def describe_runtime() -> dict[str, str]:
                                return {
                                "python": sys.version.split()[0],
                                "platform": f"{platform.system()} {platform.machine()}",
                                "executable": str(Path(sys.executable).name),
                                "cwd": str(Path.cwd()),
                                }


                                def main() -> int:
                                runtime = describe_runtime()
                                score = compute_score([1, 2, 3, 4])

                                print("Nuitka smoke test")
                                print(f"Python: {runtime['python']}")
                                print(f"Platform: {runtime['platform']}")
                                print(f"Executable: {runtime['executable']}")
                                print(f"Working directory: {runtime['cwd']}")
                                print(f"Time: {datetime.now().isoformat(timespec='seconds')}")
                                print(f"Score: {score}")

                                if score != 47:
                                print("Unexpected score")
                                return 2

                                return 0


                                if __name__ == "__main__":
                                raise SystemExit(main())
                                '@ | Set-Content -Encoding UTF8 hello_nuitka.py
```

Pourquoi ce script est utile : il teste les imports standards, l'exécution d'une fonction, le code retour, l'environnement d'exécution et la cohérence du résultat métier.

Étape 1 — exécuter le script source avec CPython

Avant toute compilation, il faut prouver que le script fonctionne normalement. Si cette étape échoue, le problème vient du code Python ou de l'environnement, pas de Nuitka.

Commande

```
python hello_nuitka.py
```

Sortie attendue

```
Nuitka smoke test
                                Python: 3.12.x
                                Platform: Windows AMD64
                                Executable: python.exe
                                Working directory: C:\dev\nuitka_quickstart
                                Time: 2026-04-26T12:34:56
                                Score: 47
```

Contrôles

Contrôle	Attendu
Le script démarre	Pas de traceback Python
La version Python s'affiche	3.10+ recommandé
La plateforme s'affiche	Windows / Linux / macOS
Le score s'affiche	`Score: 47`
Le code retour	`0`

Vérifier le code retour

```
# Linux / macOS
                                echo $?

                                # Windows CMD
                                echo %ERRORLEVEL%

                                # Windows PowerShell
                                $LASTEXITCODE
```

Validation : si le script source affiche `Score: 47` et retourne `0`, on peut passer à la compilation Nuitka.

Étape 2 — compilation accelerated / build simple

Le premier build Nuitka doit rester simple. Il valide que Nuitka démarre correctement, que la toolchain native est disponible, et qu'un exécutable peut être généré.

Commande simple

```
python -m nuitka hello_nuitka.py
```

Commande avec dossier de sortie

```
python -m nuitka \
                                --output-dir=build/nuitka \
                                hello_nuitka.py
```

Variante avec logs

```
# Linux / macOS
                                python -m nuitka --output-dir=build/nuitka hello_nuitka.py \
                                2>&1 | tee build/logs/quickstart_accelerated.log
```

Ce que ce mode produit

Élément	Description
Binaire	Exécutable généré pour la machine locale
Dossier build	Fichiers intermédiaires de compilation
Dépendances	Pas encore un livrable autonome complet
Usage	Test rapide de compatibilité et de toolchain

> **Attention :** ce mode n'est pas le format recommandé pour livrer à un client. Il sert surtout à valider rapidement que Nuitka sait compiler le script.

Exécuter le binaire accelerated

```
# Windows
                        hello_nuitka.exe

                        # Linux / macOS
                        ./hello_nuitka

                        # Si --output-dir=build/nuitka a été utilisé :
                        # Windows
                        build\nuitka\hello_nuitka.exe

                        # Linux / macOS
                        ./build/nuitka/hello_nuitka
```

Étape 3 — build standalone

Le mode `standalone` crée un dossier contenant l'exécutable et les dépendances nécessaires. C'est le format le plus important pour apprendre Nuitka, car il permet d'inspecter ce qui est réellement embarqué.

Commande standalone

```
python -m nuitka \
                                --standalone \
                                --assume-yes-for-downloads \
                                --output-dir=build/nuitka \
                                hello_nuitka.py
```

Exécution

```
# Windows
                                build\nuitka\hello_nuitka.dist\hello_nuitka.exe

                                # Linux / macOS
                                ./build/nuitka/hello_nuitka.dist/hello_nuitka
```

Anatomie du standalone

```
hello_nuitka.dist/
│
├── hello_nuitka.exe / hello_nuitka
├── python runtime components
├── standard library pieces
├── extension modules
├── native libraries
└── support files
```

Pourquoi standalone d'abord : si une DLL, un fichier data ou une dépendance manque, le dossier `.dist` permet d'inspecter et de comprendre beaucoup plus facilement.

Contrôle du dossier standalone

Contrôle	Commande	Attendu
Le dossier existe	`dir` ou `ls`	`hello_nuitka.dist`
L'exécutable existe	`dir hello_nuitka.dist`	`hello_nuitka.exe` ou binaire Linux
L'exécution fonctionne	Lancer depuis `.dist`	`Score: 47`
Code retour	`echo` variable shell	`0`

Étape 4 — build onefile

Le mode `onefile` produit un seul fichier exécutable. C'est très pratique pour distribuer un outil, mais le debug est moins direct qu'en standalone. Il faut donc construire le onefile seulement après validation du standalone.

Commande onefile

```
python -m nuitka \
    --standalone \
    --onefile \
    --assume-yes-for-downloads \
    --output-dir=build/nuitka \
    hello_nuitka.py
```

Exécution

```
# Windows
build\nuitka\hello_nuitka.exe

# Linux / macOS
./build/nuitka/hello_nuitka.bin
# ou selon la sortie :
./build/nuitka/hello_nuitka
```

Cycle de vie d'un onefile

```
onefile executable
│
├── Start
├── Prepare temporary extraction area
├── Extract embedded payload
├── Launch compiled application
├── Run program
└── Cleanup depending on runtime behavior
```

Conséquence : un onefile peut démarrer un peu moins vite qu'un standalone, car il doit préparer son environnement interne.

Quand éviter onefile ?

Situation	Pourquoi	Préférence
Debug actif	Les fichiers internes sont moins visibles	Standalone
Application avec beaucoup de data	Extraction plus lourde	Standalone
Environnement entreprise avec antivirus strict	Onefile peut être plus suspect	Standalone signé
Service serveur	La simplicité onefile apporte peu	Standalone ou module

---

Comparer source vs binaires

Après compilation, il faut comparer le comportement du script source, du binaire simple, du standalone et du onefile. La sortie ne sera pas toujours strictement identique pour les champs comme `Executable`, mais le résultat métier doit être identique.

Commandes de comparaison

```
# 1. Source
python hello_nuitka.py

# 2. Accelerated
# Windows
hello_nuitka.exe
# Linux / macOS
./hello_nuitka

# 3. Standalone
# Windows
build\nuitka\hello_nuitka.dist\hello_nuitka.exe
# Linux / macOS
./build/nuitka/hello_nuitka.dist/hello_nuitka

# 4. Onefile
# Windows
build\nuitka\hello_nuitka.exe
# Linux / macOS
./build/nuitka/hello_nuitka.bin
```

Ce qui doit être identique

Champ	Identique ?	Commentaire
`Nuitka smoke test`	Oui	L'application démarre
`Python`	En général oui	Version du runtime utilisé
`Platform`	Oui	Même OS / architecture
`Executable`	Non forcément	python.exe vs binaire compilé
`Working directory`	Dépend du lancement	Attention aux chemins relatifs
`Time`	Non	Horodatage courant
`Score`	Oui	Doit rester `47`

Script de comparaison simple

```
# Linux / macOS : capturer les sorties
                        python hello_nuitka.py > build/logs/source.out
                        ./build/nuitka/hello_nuitka.dist/hello_nuitka > build/logs/standalone.out

                        # Comparer visuellement
                        cat build/logs/source.out
                        cat build/logs/standalone.out

                        # Windows PowerShell
                        python hello_nuitka.py | Tee-Object build\logs\source.out
                        build\nuitka\hello_nuitka.dist\hello_nuitka.exe | Tee-Object
build\logs\standalone.out
```

Validation fonctionnelle : le champ `Score` doit rester à `47` et le code retour doit rester à `0`.

Comprendre les fichiers générés

Nuitka génère plusieurs fichiers et dossiers selon le mode utilisé. Les comprendre évite de supprimer le mauvais dossier ou de livrer un artefact incomplet.

Structure après plusieurs builds

```
nuitka_quickstart/
│
├── hello_nuitka.py
│
├── hello_nuitka.build/
│   └── fichiers intermédiaires C/C++
│
├── hello_nuitka.exe / hello_nuitka
│   └── build simple / accelerated
│
├── build/
│   └── nuitka/
│       ├── hello_nuitka.build/
│       ├── hello_nuitka.dist/
│       │   ├── hello_nuitka.exe
│       │   └── dépendances runtime
│       │
│       └── hello_nuitka.exe / hello_nuitka.bin
│           └── onefile
│
└── build/logs/
├── source.out
├── standalone.out
└── quickstart_build.log
```

Rôle des artefacts

Artefact	Rôle	À livrer ?
`.build/`	Fichiers intermédiaires	Non
`.dist/`	Dossier standalone complet	Oui, si mode standalone
`.onefile-build/`	Intermédiaire onefile	Non
`.exe` / binaire	Exécutable généré	Oui selon mode
`logs/`	Traçabilité	Interne seulement

Attention : en standalone, il ne faut pas livrer uniquement l'exécutable. Il faut livrer tout le dossier `.dist`.

Debug du quickstart

Si le quickstart échoue, il faut identifier rapidement la couche fautive : script Python, environnement virtuel, Nuitka, compilateur C/C++, packaging ou runtime.

Symptôme	Cause probable	Correction
`python hello_nuitka.py` échoue	Erreur dans le script ou mauvais Python	Corriger avant Nuitka
`No module named nuitka`	Nuitka non installé dans le venv actif	`python -m pip install -U nuitka`
Compilateur introuvable	MSVC / GCC / Clang absent	Installer Build Tools ou `build-essential`
Build très long	Compilation native normale ou cache absent	Installer `ccache`, patienter, garder les logs
Standalone fonctionne, onefile échoue	Extraction temporaire ou chemin runtime	Continuer en standalone pour diagnostiquer
Score différent	Bug applicatif ou mauvais fichier exécuté	Vérifier chemin, nettoyer build, relancer

Diagnostic par couche

```
Failure
│
├── Source fails?
│   └── Fix Python code
│
├── Nuitka missing?
│   └── Install in active venv
│
├── Compiler missing?
│   └── Install native toolchain
│
├── Standalone fails?
│   └── Inspect .dist folder
│
└── Onefile fails?
└── Return to standalone debug
```

Commandes utiles

```
# Vérifier Python réellement utilisé
python -c "import sys; print(sys.executable); print(sys.version)"

# Vérifier Nuitka
python -m nuitka --version

# Nettoyer les anciens builds
# Linux / macOS
rm -rf hello_nuitka.build hello_nuitka.dist build/nuitka

# Windows PowerShell
Remove-Item -Recurse -Force hello_nuitka.build, hello_nuitka.dist,
build\nuitka
```

> **Réflexe pro :** ne pas enchaîner les options au hasard. Revenir au dernier niveau qui fonctionne : source, accelerated, standalone, puis onefile.

Après le quickstart : passer à un vrai module

Une fois le quickstart validé, la suite logique est de compiler un petit module réel, par exemple un morceau isolé de `ideolab_admin_tools` : moteur de scoring, règle d'analyse, introspection de modèles, ou générateur de rapport.

Progression recommandée

```
hello_nuitka.py
|
└── OK
|
▼
small_engine.py
|
└── OK
|
▼
private_core/scoring.py
|
└── OK
|
▼
private_core/analyzer.py
|
└── OK
|
▼
integration with Django addon
```

Critères pour choisir le premier vrai module

Critère	Bon candidat	Mauvais candidat
Stabilité	Code peu modifié	Code qui change chaque jour
Dépendances	Imports simples	Imports dynamiques massifs
Django	Module métier isolé	`admin.py` complet
Tests	Fonctions testables	Logique uniquement liée à l'UI
Valeur métier	Algorithme propriétaire	Glue code sans valeur sensible

---

Mini-template pour le prochain test

```
# private_core/scoring.py

                        from __future__ import annotations


                        def compute_admin_score(model_count: int, action_count: int, filter_count: int) ->
int:
                        score = 0
                        score += model_count * 10
                        score += action_count * 5
                        score += filter_count * 3
                        return score


                        def classify_score(score: int) -> str:
                        if score >= 100:
                        return "advanced"
                        if score >= 50:
                        return "intermediate"
                        return "basic"
```

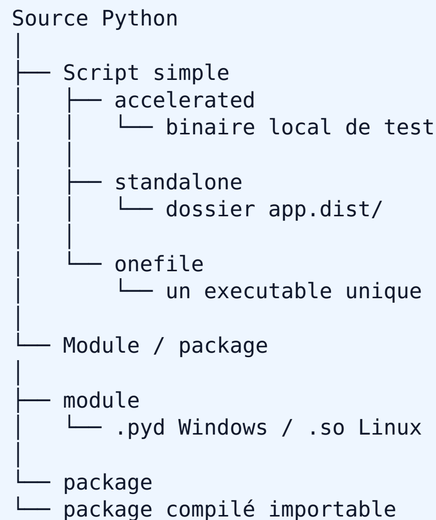
Objectif suivant : compiler ce type de module avec `--module`, puis l'importer depuis une couche Python classique.

3.1 Modes de compilation Nuitka — accelerated, standalone, onefile, module, package

Vue d'ensemble des modes Nuitka

Nuitka peut produire plusieurs types d'artefacts. Le choix du mode est stratégique : on ne choisit pas le même mode pour **tester vite**, **livrer un outil CLI**, **protéger un cœur métier**, **distribuer une application client** ou **compiler une librairie importable**.

Carte mentale des sorties



Règle simple : pour apprendre et diagnostiquer, utiliser `standalone`. Pour protéger une logique métier dans un addon Django, préférer `module`. Pour livrer à un utilisateur final, passer à `onefile` seulement après validation.

Tableau comparatif rapide

Mode	Commande moderne	Sortie	Usage principal	Difficulté
Accelerated	`--mode=accelerated`	Executable local	Premier test rapide	Faible
Standalone	`--mode=standalone`	Dossier `.dist`	Distribution contrôlable	Moyenne
Onefile	`--mode=onefile`	Un executable unique	Livraison simple	Moyenne à forte
Module	`--mode=module`	`.pyd` / `.so`	Protéger un module importable	Moyenne
Package	`--mode=module package_name`	Package compilé	Librairie propriétaire	Forte

Ancienne syntaxe vs syntaxe moderne

Nuitka accepte plusieurs écritures. Dans un guide professionnel, il est plus clair de présenter `--mode=...`, tout en mentionnant les options historiques encore très utilisées.

Objectif	Syntaxe moderne	Syntaxe équivalente courante
Build simple	`--mode=accelerated`	Aucune option spéciale : `python -m nuitka app.py`
Standalone	`--mode=standalone`	`--standalone`

Objectif	Syntaxe moderne	Syntaxe équivalente courante
Onefile	`--mode=onefile`	`--standalone --onefile`
Module importable	`--mode=module`	`--module`

Mode Accelerated — premier build local

Le mode **accelerated** est le mode de découverte. Il compile le script en un exécutable local, mais ce n'est pas encore un livrable autonome complet. Il sert surtout à vérifier que Nuitka sait analyser le code, générer le C/C++ et appeler le compilateur système.

Commande

```
# Syntaxe moderne
python -m nuitka --mode=accelerated hello_nuitka.py

# Équivalent courant
python -m nuitka hello_nuitka.py

# Avec dossier de sortie propre
python -m nuitka \
--mode=accelerated \
--output-dir=build/nuitka \
hello_nuitka.py
```

Flux de build

```
hello_nuitka.py
│
▼
Nuitka frontend
│
▼
C/C++ generated files
│
▼
Native compiler
│
▼
Local executable
```

Quand l'utiliser ?

Situation	Pertinence
Premier test Nuitka	Très bon choix
Validation toolchain C/C++	Très utile
Livraison client	Non recommandé
Application avec dépendances complexes	Insuffisant
Debug rapide	Oui

Attention : un executable accelerated peut encore dépendre fortement de l'environnement local. Il ne faut pas le considérer comme une distribution autonome.

Risques et limites

Limite	Impact	Suite recommandée
Dépendance à l'environnement local	Peut fonctionner chez vous mais pas ailleurs	Passer à standalone
Pas de dossier complet de dépendances	Diagnostic incomplet pour livraison	Tester `--mode=standalone`
Peu représentatif d'un vrai packaging	Faux sentiment de sécurité	Ne pas s'arrêter à ce mode

---

Mode Standalone — le mode de diagnostic sérieux

Le mode **standalone** produit un dossier `.dist` contenant l'exécutable et les dépendances nécessaires. C'est le mode central pour apprendre Nuitka, diagnostiquer les dépendances, préparer une livraison contrôlée et comprendre ce qui est réellement embarqué.

Commande

```
# Syntaxe moderne
python -m nuitka \
--mode=standalone \
--assume-yes-for-downloads \
--output-dir=build/nuitka \
hello_nuitka.py

# Syntaxe équivalente courante
python -m nuitka \
--standalone \
--assume-yes-for-downloads \
--output-dir=build/nuitka \
hello_nuitka.py
```

Exécution

```
# Windows
build\nuitka\hello_nuitka.dist\hello_nuitka.exe

# Linux / macOS
./build/nuitka/hello_nuitka.dist/hello_nuitka
```

Structure produite

```
hello_nuitka.dist/
│
├── hello_nuitka.exe / hello_nuitka
├── python runtime components
├── standard library fragments
├── compiled extension modules
├── native DLL / SO / DYLIB
├── package dependencies
└── included data files
```

Mode recommandé pour apprendre : en cas de fichier manquant, de DLL absente ou de problème d'import, le dossier `.dist` permet d'inspecter concrètement le livrable.

Avantages / limites

Aspect	Avantage	Limite	Action recommandée
Diagnostic	Très lisible	Beaucoup de fichiers	Inspecter le dossier `.dist`
Distribution	Assez robuste	Dossier volumineux	Zipper tout le dossier
Dépendances natives	Visibles dans le dossier	Pas toujours triviales	Tester sur machine propre
Support entreprise	Souvent plus acceptable qu'un onefile	Installation moins élégante	Créer un installateur si besoin

> **Erreur fréquente :** livrer uniquement l'exécutable contenu dans `.dist`. En standalone, il faut livrer tout le dossier `.dist`.

Mode Onefile — un seul exécutable à livrer

Le mode **onefile** produit un exécutable unique. C'est séduisant pour l'utilisateur final, mais plus délicat à diagnostiquer. Nuitka doit embarquer un payload interne, l'extraire au lancement, puis démarrer l'application.

Commande

```
# Syntaxe moderne
python -m nuitka \
--mode=onefile \
--assume-yes-for-downloads \
--output-dir=build/nuitka \
hello_nuitka.py

# Syntaxe équivalente courante
python -m nuitka \
--standalone \
--onefile \
--assume-yes-for-downloads \
--output-dir=build/nuitka \
hello_nuitka.py
```

Exécution

```
# Windows
build\nuitka\hello_nuitka.exe

# Linux / macOS
./build/nuitka/hello_nuitka.bin
# ou selon la sortie :
./build/nuitka/hello_nuitka
```

Cycle de vie onefile

```
onefile executable
│
├── Start executable
├── Prepare temporary area
├── Extract internal payload
├── Launch compiled program
├── Run application
└── Cleanup / exit
```

> **Règle d'or :** ne jamais commencer par onefile. Valider d'abord le mode standalone, puis seulement ensuite produire le onefile.

Avantages / risques

Aspect	Avantage	Risque	Conseil
Distribution	Un seul fichier	Fichier plus gros	Très pratique pour CLI client
Démarrage	Simple côté utilisateur	Extraction temporaire	Tester temps de démarrage
Debug	Moins de fichiers à gérer	Moins inspectable	Déboguer en standalone
Antivirus	Livraison compacte	Peut être plus suspect	Signature binaire si distribution pro
Fichiers data	Tout peut être embarqué	Chemins runtime plus sensibles	Éviter les chemins relatifs fragiles

> **Piège classique :** une application fonctionne en standalone mais échoue en onefile parce qu'elle cherche un fichier à côté de l'exécutable, alors qu'il est extrait ailleurs.

Mode Module — compiler un cœur métier importable

Le mode **module** est souvent le plus intéressant pour protéger une partie sensible d'une application Python ou d'un addon Django. Il produit une extension binaire importable depuis Python, comme un module natif.

Exemple simple

```
# core_engine.py contient le cœur propriétaire
                                python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka \
                                core_engine.py

                                # Syntaxe équivalente
                                python -m nuitka \
                                --module \
                                --output-dir=build/nuitka \
                                core_engine.py
```

Résultats typiques

```
# Windows
                                core_engine.cp312-win_amd64.pyd

                                # Linux
                                core_engine.cpython-312-x86_64-linux-gnu.so

                                # macOS
                                core_engine.cpython-312-darwin.so
```

Architecture recommandée

```
my_addon/
│
├── admin.py
├── views.py
├── apps.py
├── templates/
│
├── public_api.py
│   └── imports protected core
│
└── private_core/
├── scoring.py          -> compiled
├── analyzer.py         -> compiled
├── rules.py            -> compiled
└── report_builder.py   -> compiled
```

Pour un addon Django : ce mode est souvent le plus raisonnable. L'application reste Python et intégrable, mais le cœur critique devient binaire.

Module compilé : avantages et contraintes

Aspect	Avantage	Contrainte	Recommandation
Protection	Le source n'est plus livré directement	Pas inviolable	Compiler les algorithmes sensibles
Intégration	Importable depuis Python	Nom de fichier dépend de Python / OS	Builder par plateforme cible
Django	Compatible avec une app Django classique	Éviter de compiler templates / migrations	Garder une façade Python claire
Debug	Périmètre réduit	Tracebacks moins confortables	Tests unitaires avant compilation

Test d'import après compilation

```
python - <<'PY'
                        import core_engine

                        print("module:", core_engine)
                        print("result:", core_engine.compute_score([1, 2, 3]))
                        PY
```

Mode Package — compiler une librairie ou un sous-package

Le mode package consiste à compiler non pas un seul fichier isolé, mais un ensemble cohérent de modules Python. C'est puissant pour une librairie propriétaire, mais plus exigeant : imports relatifs, fichiers `__init__.py`, ressources, packaging et compatibilité doivent être soigneusement testés.

Structure exemple

```
private_core/
│
├── __init__.py
├── scoring.py
├── analyzer.py
├── rules.py
├── report_builder.py
└── resources/
└── default_rules.json
```

Commande indicative

```
python -m nuitka \
    --mode=module \
    --include-package=private_core \
    --output-dir=build/nuitka \
    private_core
```

Quand compiler un package ?

Contexte	Pertinence
Plusieurs modules très liés	Bon candidat
Librairie propriétaire interne	Très pertinent
Addon avec cœur métier isolé	Pertinent après test module simple
Package avec beaucoup d'imports dynamiques	Plus risqué
Package encore instable	À éviter au début

> **Progression recommandée :** compiler d'abord un seul module, puis deux ou trois modules, puis seulement ensuite un package complet.

Points de vigilance package

Point	Risque	Action
`__init__.py`	Imports implicites ou effets de bord	Le garder simple
Imports relatifs	Résolution différente si mal packagé	Tester depuis la racine du projet
Fichiers data	Règles JSON / templates absents	Ajouter `--include-data-files`
API publique	Couplage trop fort aux détails internes	Créer `public_api.py`
Distribution multi-OS	Extensions différentes selon OS/Python	Builder par plateforme

---

Quel mode choisir pour Django ou un addon Django ?

Pour Django, il faut distinguer **le framework**, **l'intégration**, **les templates**, **les migrations** et **le cœur métier propriétaire**. Tout compiler est rarement le meilleur premier choix.

Architecture recommandée

```
ideolab_admin_tools/
│
├── apps.py                -> Python classique
├── admin.py               -> Python classique
├── views.py               -> Python classique
├── urls.py                -> Python classique
├── templates/             -> fichiers data / HTML
├── static/                -> fichiers data / assets
│
├── public_api.py          -> façade claire
│
└── private_core/
├── scoring.py         -> module compilé
├── inspector.py       -> module compilé
├── analyzer.py        -> module compilé
└── rules.py           -> module compilé
```

Matrice Django

Élément Django	Compiler ?	Pourquoi
`models.py`	Plutôt non au début	Fortement lié à Django ORM et migrations
`admin.py`	Non au début	Glue code, registre admin, introspection
`views.py`	Non au début	Intégration web, templates, request/response
`templates/`	Non	Fichiers HTML à livrer comme data
`migrations/`	Non	Historique Django à garder lisible
`private_core/`	Oui	Cœur métier propriétaire

Mode conseillé pour IDEO-Lab

Projet	Mode conseillé	Périmètre	Raison
ideolab_admin_tools	`module`	Scoring, introspection, règles	Protéger la valeur ajoutée sans casser Django
Django Doctor	`module` puis package	Analyseurs statiques, règles d'erreurs	Algorithmes propriétaires et complexes
SRDF	`standalone` ou `module`	CLI, parsing, transport, dedup	Dépend du mode de déploiement
Migration Doctor	`module`	Classification, réparation, scoring	Protéger le moteur de diagnostic

Conclusion pratique : pour un addon Django, le meilleur premier choix est presque toujours `--mode=module` sur un noyau métier isolé, pas `--mode=onefile` sur tout le projet.

Arbre de décision : comment choisir le bon mode ?

```
Quel est l'objectif principal ?
│
├── Je veux tester Nuitka rapidement
│   └── Choisir : accelerated
│       python -m nuitka app.py
│
├── Je veux comprendre et diagnostiquer le packaging
│   └── Choisir : standalone
│       python -m nuitka --mode=standalone app.py
│
├── Je veux livrer un seul fichier à un utilisateur final
│   ├── Étape 1 : standalone
│   └── Étape 2 : onefile
│       python -m nuitka --mode=onefile app.py
│
├── Je veux protéger un algorithme propriétaire
│   └── Choisir : module
│       python -m nuitka --mode=module core_engine.py
│
├── Je veux protéger plusieurs modules liés
│   ├── Étape 1 : module simple
│   └── Étape 2 : package compilé
│
└── Je veux compiler tout Django
├── Possible dans certains cas
├── Mais déconseillé comme premier objectif
└── Extraire un private_core compilable
```

Matrice de décision rapide

Besoin	Mode	Pourquoi
Smoke test	Accelerated	Le plus rapide à lancer
Debug packaging	Standalone	Artefacts inspectables
Distribution simple	Onefile	Un seul fichier
Protection code	Module	Extension binaire importable
Librairie propriétaire	Package	Plusieurs modules protégés
Addon Django	Module	Façade Django conservée

Signaux d'alerte

- Vous démarrez directement avec `onefile` sans avoir testé standalone.
- Vous essayez de compiler tout Django dès le premier jour.
- Le module ciblé n'a aucun test unitaire.
- Le code utilise beaucoup d'imports dynamiques non maîtrisés.
- Le module lit des fichiers par chemins relatifs fragiles.
- Vous ne savez pas quel Python ou quel compilateur est utilisé.

Décision saine : si le périmètre est flou, ne compilez pas encore. Isolez d'abord un petit noyau stable, testable et à forte valeur métier.

Cheat-sheet des modes Nuitka

Commandes essentielles

```
# 1. Build simple / accelerated
                                python -m nuitka app.py

                                # 2. Standalone
                                python -m nuitka \
                                --mode=standalone \
                                --assume-yes-for-downloads \
                                --output-dir=build/nuitka \
                                app.py

                                # 3. Onefile
                                python -m nuitka \
                                --mode=onefile \
                                --assume-yes-for-downloads \
                                --output-dir=build/nuitka \
                                app.py

                                # 4. Module compilé
                                python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka \
                                core_engine.py

                                # 5. Package / dépendances explicites
                                python -m nuitka \
                                --mode=module \
                                --include-package=private_core \
                                --output-dir=build/nuitka \
                                private_core
```

Résumé ultra-court

Mode	À retenir
Accelerated	Premier test, pas une vraie distribution.
Standalone	Meilleur mode pour diagnostiquer et livrer un dossier complet.
Onefile	Pratique, mais à faire après standalone.
Module	Meilleur choix pour protéger un moteur propriétaire.
Package	Puissant, mais à réserver après maîtrise du mode module.

Pour ton cas IDEO-Lab : commencer par `--mode=module` sur un fichier du type `private_core/scoring.py`, puis élargir progressivement vers un sous-package compilé.

Ordre professionnel recommandé

```
1. accelerated
└── valider Nuitka + compilateur

2. standalone
└── valider packaging et dépendances

3. module
└── protéger un cœur métier isolé

4. package
└── protéger un ensemble de modules

5. onefile
└── livrer simplement, seulement si standalone OK
```

3.2 Options essentielles Nuitka — includes, excludes, plugins, data files, reports, jobs

Pourquoi les options Nuitka sont importantes ?

Nuitka fonctionne très bien sur des scripts simples, mais les vrais projets Python contiennent des imports dynamiques, des fichiers JSON/YAML, des templates, des dépendances natives, des plugins, des assets, des dossiers de sortie, des logs et des contraintes de performance. Les options servent à rendre le build **explicite**, **reproductible** et **diagnostiquable**.

Carte des familles d'options

```
Nuitka options
│
├── Build mode
│   ├── --mode=accelerated
│   ├── --mode=standalone
│   ├── --mode=onefile
│   └── --mode=module
│
├── Imports
│   ├── --include-module
│   ├── --include-package
│   ├── --nofollow-import-to
│   └── --noinclude-*
│
├── Data files
│   ├── --include-data-files
│   ├── --include-data-dir
│   └── package data rules
│
├── Plugins
│   ├── --enable-plugin
│   ├── --disable-plugin
│   └── framework helpers
│
├── Output / reports
│   ├── --output-dir
│   ├── --remove-output
│   ├── --report
│   └── --show-progress
│
└── Performance
├── --jobs
├── --lto
└── compiler cache
```

Tableau de synthèse

Famille	But	Exemple	Risque si oublié
Output	Organiser les artefacts	`--output-dir=build/nuitka`	Projet pollué, builds confus
Imports	Inclure modules dynamiques	`--include-module=x.y`	`ModuleNotFoundError`
Data	Inclure fichiers non Python	`--include-data-files`	`FileNotFoundError`
Plugins	Support frameworks/packages	`--enable-plugin=tk-inter`	Framework incomplet au runtime
Report	Analyser le build	`--report=report.xml`	Diagnostic difficile
Jobs	Accélérer compilation	`--jobs=8`	Build lent

Règle pro : dès qu'un build marche, figer la commande dans un script. Une commande Nuitka importante ne doit pas rester uniquement dans l'historique du terminal.

Options de build générales

Ces options contrôlent la forme du build, le nettoyage, le dossier de sortie, les téléchargements automatiques, la progression et la verbosité.

Option	Usage	Quand l'utiliser	Remarque
`--mode=accelerated`	Build simple local	Premier test	Pas un livrable autonome
`--mode=standalone`	Dossier autonome `.dist`	Diagnostic sérieux	Meilleur mode avant onefile
`--mode=onefile`	Un seul exécutable	Livraison finale	À faire après standalone
`--mode=module`	Extension importable	Protection d'un moteur	Idéal pour addon Django
`--output-dir=build/nuitka`	Centralise les sorties	Toujours en projet pro	Évite de polluer la racine
`--remove-output`	Supprime anciens outputs	Build propre	Attention si logs dans même dossier
`--assume-yes-for-downloads`	Accepte certains téléchargements nécessaires	CI / build non interactif	Utile sur Windows / dépendances
`--show-progress`	Affiche progression	Build long	Confort développeur
`--show-memory`	Affiche usage mémoire	Serveur limité	Utile sur gros projets

Commande propre de base

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --assume-yes-for-downloads \
                                hello_nuitka.py
```

Organisation recommandée

```
project/
│
├── src/
├── scripts/
│   └── build_nuitka.py
│
├── build/
│   ├── nuitka/
│   ├── logs/
│   └── reports/
│
└── dist/
└── release archives
```

Imports : include, exclude, nofollow

Nuitka suit très bien les imports explicites. Les problèmes apparaissent avec les imports dynamiques, les plugins chargés par nom, les modules optionnels, les frameworks qui découvrent automatiquement des fichiers, ou les paquets volontairement non utilisés en production.

Imports explicites vs dynamiques

```
Import explicite
│
├── import package.module
└── from package import module

Nuitka peut suivre facilement.
│
▼
Import dynamique
│
├── importlib.import_module(name)
├── __import__(name)
├── plugin registry
└── string path in settings

Nuitka peut avoir besoin d'une option.
```

Options principales

Option	Usage
`--include-module=name`	Inclure un module précis chargé dynamiquement.
`--include-package=pkg`	Inclure tout un package.
`--nofollow-import-to=pkg`	Ne pas suivre les imports vers un package.
`--noinclude-*`	Exclure certaines familles selon les options disponibles.
`--follow-imports`	Suivre plus largement les imports du projet.

Exemples

Inclure un module dynamique

```
python -m nuitka \
                                --mode=standalone \
                                --include-module=myapp.plugins.csv_exporter \
                                app.py
```

Inclure un package complet

```
python -m nuitka \
                                --mode=standalone \
                                --include-package=myapp.plugins \
                                app.py
```

Exclure un package lourd inutile

```
python -m nuitka \
                                --mode=standalone \
                                --nofollow-import-to=tests \
                                --nofollow-import-to=matplotlib \
                                app.py
```

Cas plugin registry

```
# Code Python
                                plugin = importlib.import_module(settings.EXPORT_PLUGIN)

                                # Build
                                python -m nuitka \
                                --include-module=myapp.exporters.pdf \
                                --include-module=myapp.exporters.csv \
                                app.py
```

> **Conseil :** si un module est chargé par une chaîne de caractères, il faut souvent l'inclure explicitement. C'est très fréquent dans les systèmes de plugins.

Data files : fichiers non Python à embarquer

Nuitka compile le code, mais une application réelle dépend souvent de fichiers non Python : templates HTML, fichiers JSON, YAML, TOML, certificats, images, règles métier, dictionnaires, fichiers SQL, modèles ML, etc. Ces fichiers doivent être pensés comme des dépendances runtime.

Carte des fichiers data

```
                                Application runtime
                                │
                                ├── Python code
                                │   └── compiled by Nuitka
                                │
                                └── Data files
                                ├── templates/*.html
                                ├── static/css/*.css
                                ├── static/js/*.js
                                ├── rules/*.json
                                ├── config/*.yaml
                                ├── certificates/*.pem
                                ├── sql/*.sql
                                └── assets/images/*
```

> **Erreur classique :** le programme fonctionne depuis le code source, mais le binaire échoue car un fichier JSON, YAML ou template n'a pas été inclus.

Options principales

Option	Usage
`--include-data-files=src=dst`	Inclure un ou plusieurs fichiers.
`--include-data-dir=src=dst`	Inclure un dossier complet.
`--include-package-data=pkg`	Inclure les data files d'un package.

Option	Usage
`--noinclude-data-files=pattern`	Exclure certains fichiers data.

Exemples concrets

Inclure un fichier YAML

```
python -m nuitka \
                                --mode=standalone \
                                --include-data-files=config/default.yaml=config/default.yaml \
                                app.py
```

Inclure un dossier de règles

```
python -m nuitka \
                                --mode=standalone \
                                --include-data-dir=rules=rules \
                                app.py
```

Inclure templates et static

```
python -m nuitka \
                                --mode=standalone \
                                --include-data-dir=templates=templates \
                                --include-data-dir=static=static \
                                app.py
```

Exclure des fichiers inutiles

```
python -m nuitka \
                                --mode=standalone \
                                --include-data-dir=resources=resources \
                                --noinclude-data-files=resources/dev_only/* \
                                app.py
```

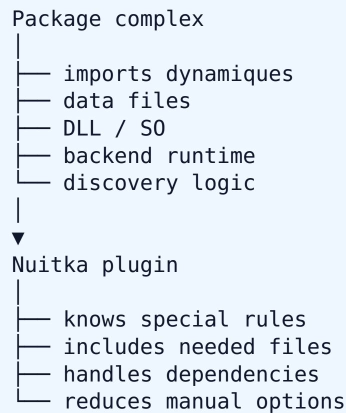
Table de contrôle data files

Type	Exemple	Risque	Test à faire
Template	`templates/report.html`	Template introuvable	Générer un rapport depuis le binaire
JSON rules	`rules/default.json`	Règles absentes	Charger toutes les règles
YAML config	`config/default.yaml`	Config absente	Lancer avec config par défaut
Certificat	`certs/ca.pem`	Erreur TLS	Tester connexion réseau
Static	`static/app.css`	UI cassée	Ouvrir l'interface finale

Plugins Nuitka : support framework et packages spéciaux

Certains packages Python nécessitent des règles particulières : fichiers data, bibliothèques natives, imports dynamiques, backends graphiques, frameworks ou packages scientifiques. Les plugins Nuitka aident à gérer ces cas.

Principe



Options plugins

Option	Usage
`--enable-plugin=name`	Activer un plugin Nuitka.
`--disable-plugin=name`	Désactiver un plugin.
`--plugin-list`	Lister les plugins disponibles.
`--user-plugin=file.py`	Charger un plugin utilisateur avancé.

```
# Lister les plugins disponibles
python -m nuitka --plugin-list

# Exemple : activer un plugin
python -m nuitka \
--mode=standalone \
--enable-plugin=tk-inter \
app.py
```

Quand penser aux plugins ?

Symptôme	Cause possible	Action
Package graphique incomplet	Backend / data non inclus	Vérifier plugin associé
Framework charge des composants par découverte	Imports dynamiques	Plugin ou include explicite
Warnings Nuitka mentionnant plugin	Support spécial recommandé	Lire warning et activer si nécessaire
Runtime OK en source mais KO en binaire	Dépendance spéciale absente	Tester plugin + standalone

> **Conseil :** ne pas activer tous les plugins au hasard. Activer uniquement ceux qui correspondent à vos dépendances réelles.

Output, logs, rapports et traçabilité

Un build professionnel doit être traçable. Il faut pouvoir savoir quelle commande a été lancée, avec quelle version de Python, quelle version de Nuitka, quelles options, quels modules inclus, et quels warnings ont été produits.

Options utiles

Option	But
`--output-dir=build/nuitka`	Centraliser les sorties.
`--report=build/reports/report.xml`	Produire un rapport XML de compilation.
`--show-progress`	Afficher la progression du build.
`--show-memory`	Afficher l'usage mémoire pendant le build.
`--remove-output`	Nettoyer les sorties précédentes.

Structure de build pro

```
build/
│
├── nuitka/
│   ├── app.build/
│   ├── app.dist/
│   └── app.onefile-build/
│
├── logs/
│   ├── build.log
│   ├── runtime.log
│   └── smoke_test.log
│
└── reports/
├── compilation-report.xml
├── environment.txt
└── command.txt
```

Capturer un log de build

Linux / macOS

```
mkdir -p build/logs build/reports

python -m nuitka \
--mode=standalone \
--output-dir=build/nuitka \
--report=build/reports/compilation-report.xml \
--show-progress \
app.py \
2>&1 | tee build/logs/build.log
```

Windows PowerShell

```
New-Item -ItemType Directory -Force build\logs, build\reports

python -m nuitka `
--mode=standalone `
--output-dir=build\nuitka `
--report=build\reports\compilation-report.xml `
--show-progress `
app.py *>&1 | Tee-Object build\logs\build.log
```

Bonne pratique : joindre `build.log`, `command.txt`, `environment.txt` et `compilation-report.xml` à tout diagnostic sérieux.

Performance de build : jobs, LTO, cache

Nuitka peut prendre du temps sur des projets réels. Il y a deux performances différentes : la performance du **build** et la performance de **l'application compilée**. Les options ci-dessous concernent principalement le temps de compilation.

Option / outil	Effet	Avantage	Risque / coût
`--jobs=8`	Compilation parallèle	Build plus rapide	Plus de CPU / RAM
`--lto=yes`	Link Time Optimization	Optimisation plus poussée	Build plus lent, parfois plus fragile
`ccache`	Cache compilateur	Rebuilds plus rapides	Configuration dépend OS
`--show-progress`	Feedback progression	Meilleure visibilité	Pas d'accélération réelle
`--show-memory`	Suivi mémoire	Diagnostic machine limitée	Sortie plus verbeuse

Build rapide raisonnable

```
python -m nuitka \
                                --mode=standalone \
                                --jobs=8 \
                                --show-progress \
                                --output-dir=build/nuitka \
                                app.py
```

Build optimisé plus lourd

```
python -m nuitka \
                                --mode=standalone \
                                --jobs=8 \
                                --lto=yes \
                                --show-progress \
                                --show-memory \
                                --output-dir=build/nuitka \
                                app.py
```

Conseil : ne pas activer `--lto=yes` dès le début. Stabiliser d'abord le build, puis mesurer si LTO apporte réellement quelque chose.

Options et stratégie de protection

Nuitka n'est pas une solution magique d'inviolabilité, mais certaines options et décisions d'architecture permettent d'améliorer nettement la protection pratique du code source : compiler le bon périmètre, exclure les tests, éviter d'inclure des fichiers sensibles, et livrer uniquement ce qui est nécessaire.

Ce qu'il faut compiler

```
Good target for protection
│
├── private_core/scoring.py
├── private_core/analyzer.py
├── private_core/rules_engine.py
├── private_core/parser.py
└── private_core/report_builder.py

Poor target at first
│
├── tests/
├── migrations/
├── templates/
├── admin.py glue code
└── unstable experimental files
```

Options utiles indirectement

Option	Impact protection
`--mode=module`	Transforme un module métier en extension binaire.
`--nofollow-import-to=tests`	Évite d'embarquer du code de test inutile.
`--noinclude-data-files=...`	Évite d'embarquer des fichiers sensibles non nécessaires.
`--include-package`	Permet d'inclure uniquement le bon périmètre.
`--output-dir`	Sépare clairement artefacts livrables et sources.

À ne jamais embarquer par erreur

Fichier / dossier	Risque	Action
`.env`	Secrets, tokens, mots de passe	Ne jamais inclure
`.git/`	Historique source complet	Exclure totalement
`tests/`	Révèle cas internes et logique attendue	Exclure du livrable
`docs/internal/`	Architecture privée	Ne pas livrer
`dev_config.yaml`	Chemins internes / secrets	Remplacer par config publique

> **Important :** compiler ne compense pas une mauvaise hygiène de livraison. Un build qui embarque `.env`, tests internes ou documentation privée reste dangereux.

Options utiles pour un addon Django ou IDEO-Lab

Dans un addon Django, le plus raisonnable est souvent de compiler le cœur métier sous forme de module, puis de garder l'intégration Django en Python classique. Les options importantes concernent surtout les imports, les data files et l'organisation du build.

Architecture cible

```
ideolab_admin_tools/
│
├── admin.py              -> Python lisible
├── apps.py               -> Python lisible
├── views.py              -> Python lisible
├── templates/            -> data files
├── static/               -> data files
│
├── public_api.py         -> façade
│
└── private_core/
├── scoring.py        -> compile module
├── inspector.py      -> compile module
├── analyzer.py       -> compile module
└── rules.json        -> include data
```

Commande module pour cœur métier

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                ideolab_admin_tools/private_core/scoring.py
```

Commande standalone pour CLI d'analyse

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --include-data-dir=ideolab_admin_tools/templates=ideolab_admin_tools/
templates \
                                --include-data-dir=ideolab_admin_tools/static=ideolab_admin_tools/static \
                                tools/admin_scan_cli.py
```

Table de décision options Django

Besoin	Option	Commentaire
Compiler un moteur propriétaire	`--mode=module`	Meilleur premier choix
Inclure règles JSON	`--include-data-files`	Si le moteur lit des règles externes
Inclure templates pour CLI	`--include-data-dir=templates=templates`	Utile si génération HTML/PDF
Éviter tests internes	`--nofollow-import-to=tests`	Ne pas livrer le laboratoire interne
Rapport build	`--report=...`	Indispensable pour debug pro
Build reproductible	Script dédié	Éviter les commandes manuelles longues

Pour IDEO-Lab : les options les plus importantes seront probablement `--mode=module`, `--output-dir`, `--report`, `--include-data-files` pour les règles, et parfois `--include-module` si un moteur charge des plugins dynamiquement.

Commandes complètes prêtes à copier

Build standalone propre

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --jobs=8 \
                                --show-progress \
                                --show-memory \
                                --report=build/reports/compilation-report.xml \
                                --assume-yes-for-downloads \
                                hello_nuitka.py
```

Build onefile propre

```
python -m nuitka \
                                --mode=onefile \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --jobs=8 \
                                --show-progress \
                                --report=build/reports/onefile-report.xml \
                                --assume-yes-for-downloads \
                                hello_nuitka.py
```

Build module protégé

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/module-report.xml \
                                private_core/scoring.py
```

Build avec data files

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --include-data-dir=templates=templates \
                                --include-data-dir=static=static \
                                --include-data-files=config/default.yaml=config/default.yaml \
                                --include-data-files=rules/*.json=rules/ \
                                --report=build/reports/data-report.xml \
                                app.py
```

Commande à éviter au début

```
# Mauvais réflexe : trop d'options, onefile direct, pas de rapport, pas de standalone validé
                        python -m nuitka --onefile app.py
```

Approche professionnelle : commencer par une commande lisible, ajouter les options une par une, garder un rapport, puis figer le tout dans un script de build.

Mini-script de build recommandé

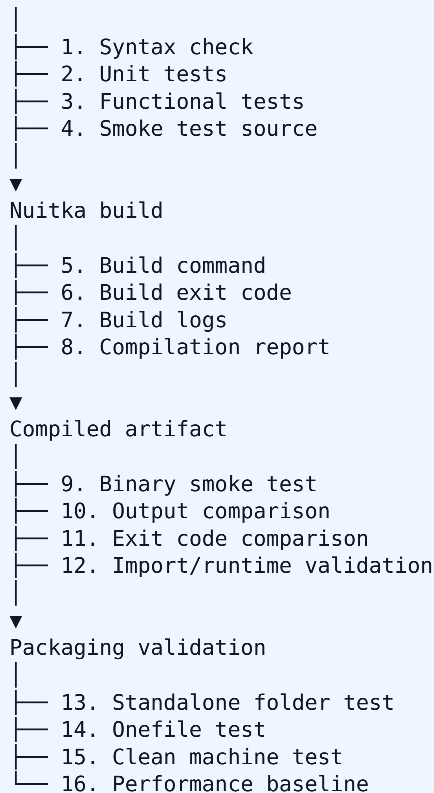
```
# scripts/build_standalone.sh
                        #!/usr/bin/env bash
                        set -euo pipefail

                        mkdir -p build/nuitka build/logs build/reports

                        python -m nuitka \
                        --mode=standalone \
                        --output-dir=build/nuitka \
                        --remove-output \
                        --jobs=8 \
                        --show-progress \
                        --report=build/reports/compilation-report.xml \
                        --assume-yes-for-downloads \
                        hello_nuitka.py \
                        2>&1 | tee build/logs/build.log
```

4.1 Tests & validation Nuitka — source vs binaire, smoke tests, pytest, performance, packaging

Objectif : prouver que le binaire se comporte comme le source

Compiler avec Nuitka ne suffit pas. Un build réussi signifie seulement que Nuitka et le compilateur C/C++ ont produit un artefact. La vraie validation consiste à prouver que le binaire produit le même comportement que le script source : mêmes résultats métier, mêmes codes retour, mêmes imports nécessaires, mêmes fichiers data disponibles, et fonctionnement sur une machine cible sans environnement de développement.

Chaîne de validation complète



```
Source Python
                                │
                                ├── 1. Syntax check
                                ├── 2. Unit tests
                                ├── 3. Functional tests
                                ├── 4. Smoke test source
                                │
                                ▼
                                Nuitka build
                                │
                                ├── 5. Build command
                                ├── 6. Build exit code
                                ├── 7. Build logs
                                ├── 8. Compilation report
                                │
                                ▼
                                Compiled artifact
                                │
                                ├── 9. Binary smoke test
                                ├── 10. Output comparison
                                ├── 11. Exit code comparison
                                ├── 12. Import/runtime validation
                                │
                                ▼
                                Packaging validation
                                │
                                ├── 13. Standalone folder test
                                ├── 14. Onefile test
                                ├── 15. Clean machine test
                                └── 16. Performance baseline
```

Ce qu'il faut comparer

Élément	Source	Binaire	Critère
Code retour	`0`	`0`	Identique
Sortie métier	`Score: 47`	`Score: 47`	Identique
stderr	Aucune erreur	Aucune erreur	Identique ou expliqué
Imports	OK	OK	Aucun module manquant
Fichiers data	OK	OK	Aucun fichier absent
Machine cible	Non applicable	OK	Démarre hors venv

Règle professionnelle : un binaire Nuitka n'est validé que s'il passe un test fonctionnel hors dossier source, idéalement sur une machine propre, une VM ou un conteneur minimal.

Matrice minimale de validation GO / NO-GO

Cette matrice sert à décider si le build peut être livré, rejeté ou investigué. Elle doit être utilisée après chaque modification importante de code, d'options Nuitka, de version Python ou de dépendances.

Niveau	Test	Commande / action	Critère GO	NO-GO si...
Source	Exécution directe	`python app.py`	Fonctionne avant compilation	Traceback ou mauvais résultat
Unitaires	Tests métier	`pytest`	Tests verts	Un test échoue
Syntaxe	Compileall	`python -m compileall .`	Aucune erreur	Erreur de syntaxe
Build	Compilation Nuitka	`python -m nuitka ...`	Exit code 0	Erreur Nuitka ou compilateur
Rapport	Report XML / logs	`--report=...`	Rapport généré	Rapport absent ou logs incomplets
Binaire	Exécution artefact	Lancer exe / binaire	Même comportement	Crash, stderr, code retour différent
Standalone	Dossier `.dist`	Lancer depuis `.dist`	Démarre correctement	DLL / fichier manquant
Onefile	Exécutable unique	Lancer onefile	Même résultat que standalone	Crash extraction ou chemin
Machine cible	Copie hors dev	Tester sans venv	Démarre sans source	Dépendance locale cachée
Performance	Baseline	Mesure source vs binaire	Pas de régression bloquante	Démarrage trop lent ou mémoire excessive

Attention : un build qui compile mais ne passe pas les tests runtime est un build cassé. La compilation n'est pas une validation fonctionnelle.

Validation du code source avant compilation

Avant de lancer Nuitka, il faut s'assurer que le code fonctionne parfaitement en Python pur. Cela évite de confondre une erreur applicative avec une erreur de compilation.

Commandes source

```
# Exécution directe
python hello_nuitka.py

# Vérifier le code retour Linux / macOS
echo $?

# Vérifier le code retour Windows CMD
echo %ERRORLEVEL%

# Vérifier le code retour Windows PowerShell
$LASTEXITCODE

# Vérification syntaxe sur un dossier
python -m compileall .
```

Contrôles attendus

Contrôle	GO	NO-GO
Traceback	Aucun	Erreur Python visible
Code retour	`0`	Différent de `0`
Résultat métier	Résultat attendu	Résultat incohérent
Warnings	Compris / acceptés	Warnings nouveaux non expliqués
Imports	Tous OK	ImportError / ModuleNotFoundError

Mini smoke test source

```
python - <<'PY'
import subprocess
import sys

cmd = [sys.executable, "hello_nuitka.py"]
p = subprocess.run(cmd, text=True, capture_output=True)

print("returncode:", p.returncode)
print("stdout:")
print(p.stdout)
print("stderr:")
print(p.stderr)

if p.returncode != 0:
raise SystemExit("Source execution failed")

if "Score: 47" not in p.stdout:
raise SystemExit("Expected business output not found")

print("OK - source smoke test passed")
PY
```

Validation du build Nuitka

La validation du build ne se limite pas à voir un fichier généré. Il faut contrôler le code retour de Nuitka, conserver les logs, produire un rapport, et vérifier que les artefacts attendus existent.

Commande de build traçable

```
mkdir -p build/logs build/reports build/nuitka

                                python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/compilation-report.xml \
                                --assume-yes-for-downloads \
                                hello_nuitka.py \
                                2>&1 | tee build/logs/build.log
```

Contrôles build

Contrôle	Commande	Attendu
Exit code build	`echo $?`	`0`
Log présent	`ls build/logs`	`build.log`
Rapport présent	`ls build/reports`	`compilation-report.xml`
Dossier dist	`ls build/nuitka`	`hello_nuitka.dist`
Executable	`find build/nuitka -name "hello*"`	Binaire présent

Build validé : code retour `0`, rapport généré, log conservé, artefact présent, puis exécution runtime réussie.

Validation du binaire généré

Un binaire doit être lancé depuis son vrai emplacement final. Pour standalone, il faut exécuter le programme depuis le dossier `.dist`. Pour onefile, il faut lancer le fichier unique et vérifier que l'extraction runtime ne casse rien.

Commandes d'exécution

```
# Source
                                python hello_nuitka.py

                                # Standalone Windows
                                build\nuitka\hello_nuitka.dist\hello_nuitka.exe

                                # Standalone Linux / macOS
                                ./build/nuitka/hello_nuitka.dist/hello_nuitka

                                # Onefile Windows
                                build\nuitka\hello_nuitka.exe

                                # Onefile Linux / macOS
                                ./build/nuitka/hello_nuitka.bin
```

Ce qu'il faut observer

Signal	GO	NO-GO
stdout	Sortie attendue	Sortie tronquée ou incohérente
stderr	Vide ou warnings connus	Traceback, DLL error

Signal	GO	NO-GO
Exit code	`0`	Différent de `0`
Résultat métier	Identique au source	Différent
Temps de démarrage	Acceptable	Dégradation excessive

Smoke test runtime automatique

```
python - <<'PY'
                        import subprocess
                        import sys
                        from pathlib import Path

                        if sys.platform == "win32":
                        binary = Path("build/nuitka/hello_nuitka.dist/hello_nuitka.exe")
                        else:
                        binary = Path("build/nuitka/hello_nuitka.dist/hello_nuitka")

                        if not binary.exists():
                        raise SystemExit(f"Binary not found: {binary}")

                        p = subprocess.run([str(binary)], text=True, capture_output=True)

                        print("returncode:", p.returncode)
                        print("stdout:", p.stdout)
                        print("stderr:", p.stderr)

                        if p.returncode != 0:
                        raise SystemExit("Binary failed")

                        if "Score: 47" not in p.stdout:
                        raise SystemExit("Business output mismatch")

                        print("OK - binary smoke test passed")
                        PY
```

Script de comparaison source vs binaire

Ce script compare l'exécution du source Python et du binaire Nuitka. Il vérifie le code retour, la sortie métier et permet de capturer les différences.

```
from __future__ import annotations

                        import difflib
                        import subprocess
                        import sys
                        from pathlib import Path


                        def run(cmd: list[str]) -> tuple[int, str, str]:
                        process = subprocess.run(cmd, text=True, capture_output=True)
                        return process.returncode, process.stdout.strip(), process.stderr.strip()


                        def normalize_output(text: str) -> list[str]:
                        """
                        Remove volatile lines from comparison.
                        Time, executable path and working directory can differ between source and binary.
                        """
                        ignored_prefixes = (
                        "Time:",
                        "Executable:",
                        "Working directory:",
                        )
                        lines = []
                        for line in text.splitlines():
                        if not line.startswith(ignored_prefixes):
                        lines.append(line)
                        return lines


                        def main() -> int:
                        source_cmd = [sys.executable, "hello_nuitka.py"]

                        if sys.platform == "win32":
                        binary_path = Path("build/nuitka/hello_nuitka.dist/hello_nuitka.exe")
                        else:
                        binary_path = Path("build/nuitka/hello_nuitka.dist/hello_nuitka")

                        binary_cmd = [str(binary_path)]

                        if not binary_path.exists():
                        print(f"Binary not found: {binary_path}", file=sys.stderr)
                        return 10

                        source_code, source_out, source_err = run(source_cmd)
                        binary_code, binary_out, binary_err = run(binary_cmd)

                        if source_code != binary_code:
                        print(f"Exit code mismatch: source={source_code}, binary={binary_code}")
                        print("SOURCE STDERR:", source_err)
                        print("BINARY STDERR:", binary_err)
                        return 20

                        if "Score: 47" not in source_out:
                        print("Expected source output not found")
                        return 30

                        if "Score: 47" not in binary_out:
                        print("Expected binary output not found")
                        return 40

                        source_lines = normalize_output(source_out)
                        binary_lines = normalize_output(binary_out)

                        if source_lines != binary_lines:
                        print("Output differs:")
                        for line in difflib.unified_diff(
                        source_lines,
                        binary_lines,
                        fromfile="source",
                        tofile="binary",
                        lineterm="",
                        ):
```

```
                         print(line)
                         return 50

                         print("OK - source and binary are coherent")
                         return 0


                         if __name__ == "__main__":
                         raise SystemExit(main())
```

Utilisation

```
# Enregistrer sous compare_source_binary.py
                                python compare_source_binary.py

                                # Code retour attendu
                                # 0 = OK
```

Pourquoi normaliser ?

Certaines lignes changent naturellement entre source et binaire : nom de l'exécutable, dossier courant, horodatage, chemin temporaire onefile. Le test doit comparer le comportement métier, pas les valeurs volatiles.

Intégrer la validation dans pytest

Pour un projet professionnel, il est préférable d'automatiser la validation. `pytest` peut lancer le script source, le binaire compilé, comparer les sorties, et échouer clairement si le comportement diverge.

Test pytest minimal

```
from __future__ import annotations

                                import subprocess
                                import sys
                                from pathlib import Path


                                def run_command(command: list[str]) -> subprocess.CompletedProcess[str]:
                                return subprocess.run(
                                command,
                                text=True,
                                capture_output=True,
                                check=False,
                                )


                                def test_source_hello_nuitka():
                                result = run_command([sys.executable, "hello_nuitka.py"])

                                assert result.returncode == 0
                                assert "Nuitka smoke test" in result.stdout
                                assert "Score: 47" in result.stdout
                                assert result.stderr == ""


                                def test_compiled_binary_hello_nuitka():
                                if sys.platform == "win32":
                                binary = Path("build/nuitka/hello_nuitka.dist/hello_nuitka.exe")
                                else:
                                binary = Path("build/nuitka/hello_nuitka.dist/hello_nuitka")

                                assert binary.exists(), f"Missing binary: {binary}"

                                result = run_command([str(binary)])

                                assert result.returncode == 0
                                assert "Nuitka smoke test" in result.stdout
                                assert "Score: 47" in result.stdout
```

Organisation recommandée

```
tests/
                                │
                                ├── test_source_behavior.py
                                ├── test_binary_behavior.py
                                ├── test_packaging.py
                                └── test_performance_baseline.py

                                build/
                                │
                                ├── nuitka/
                                ├── logs/
                                └── reports/
```

Commandes

```
# Installer pytest si nécessaire
                                python -m pip install pytest

                                # Lancer tous les tests
                                python -m pytest -v

                                # Lancer seulement les tests binaires
                                python -m pytest -v tests/test_binary_behavior.py
```

> **Bon niveau pro :** un build Nuitka devrait être accompagné d'au moins un test pytest qui lance réellement l'artefact compilé.

Mesurer les performances sans se tromper

Nuitka peut améliorer certains cas, mais ce n'est pas un accélérateur miracle. Les gains dépendent du type de code : calcul Python pur, boucles, I/O, imports lourds, bibliothèques C déjà optimisées, démarrage, packaging onefile, etc.

Mesures simples

```
# Linux / macOS : temps détaillé
                                /usr/bin/time -v python hello_nuitka.py
                                /usr/bin/time -v ./build/nuitka/hello_nuitka.dist/hello_nuitka

                                # Windows PowerShell
                                Measure-Command { python .\hello_nuitka.py }
                                Measure-Command { .\build\nuitka\hello_nuitka.dist\hello_nuitka.exe }
```

Ce qu'il faut mesurer

Métrique	Pourquoi	Attention
Temps de démarrage	Important pour CLI	Onefile peut être plus lent au démarrage
Temps total	Mesure globale	Inclut I/O, imports, environnement
CPU	Calcul pur	Bibliothèques C sont déjà optimisées
Mémoire	Serveur / agent local	Comparer plusieurs runs
Taille livrable	Distribution	Standalone plus volumineux

Micro-benchmark contrôlé

```
from __future__ import annotations

                        import statistics
                        import subprocess
                        import sys
                        import time
                        from pathlib import Path


                        def measure(command: list[str], runs: int = 5) -> list[float]:
                        values = []
                        for _ in range(runs):
                        start = time.perf_counter()
                        result = subprocess.run(command, text=True, capture_output=True)
                        end = time.perf_counter()

                        if result.returncode != 0:
                        raise RuntimeError(result.stderr)

                        values.append(end - start)
                        return values


                        def main() -> int:
                        source_cmd = [sys.executable, "hello_nuitka.py"]

                        if sys.platform == "win32":
                        binary_cmd = [str(Path("build/nuitka/hello_nuitka.dist/hello_nuitka.exe"))]
                        else:
                        binary_cmd = [str(Path("build/nuitka/hello_nuitka.dist/hello_nuitka"))]

                        source_times = measure(source_cmd)
                        binary_times = measure(binary_cmd)

                        print("source:", source_times, "avg:", statistics.mean(source_times))
                        print("binary:", binary_times, "avg:", statistics.mean(binary_times))

                        return 0


                        if __name__ == "__main__":
                        raise SystemExit(main())
```

Ne pas conclure trop vite : mesurer plusieurs runs, distinguer démarrage et calcul, et ne pas vendre Nuitka comme une garantie de performance.

Validation packaging : fichiers, imports, standalone, machine cible

Le packaging est souvent la vraie difficulté : un binaire peut fonctionner dans le dossier source, puis échouer ailleurs parce qu'il dépend implicitement d'un fichier local, d'un venv, d'une DLL, d'un package ou d'un chemin relatif.

Test hors dossier source

```
# Linux / macOS
                                mkdir -p /tmp/nuitka_runtime_test
                                cp -r build/nuitka/hello_nuitka.dist /tmp/nuitka_runtime_test/
                                cd /tmp/nuitka_runtime_test/hello_nuitka.dist
                                ./hello_nuitka

                                # Windows PowerShell
                                New-Item -ItemType Directory -Force C:\temp\nuitka_runtime_test
                                Copy-Item -Recurse build\nuitka\hello_nuitka.dist C:
\temp\nuitka_runtime_test\
                                cd C:\temp\nuitka_runtime_test\hello_nuitka.dist
                                .\hello_nuitka.exe
```

Checklist packaging

Contrôle	Pourquoi	GO
Exécution hors source	Détecte chemins cachés	OK
Sans venv actif	Détecte dépendance locale	OK
Dossier `.dist` complet	Standalone nécessite tout le dossier	OK
Data files présents	Templates / JSON / YAML	OK
DLL / SO présents	Dépendances natives	OK
Onefile testé séparément	Extraction runtime	OK

Erreurs packaging fréquentes

Erreur	Cause probable	Correction
`FileNotFoundError`	Fichier data non inclus	`--include-data-files` ou `--include-data-dir`
`ModuleNotFoundError`	Import dynamique absent	`--include-module` ou `--include-package`
`DLL load failed`	Librairie native absente	Tester standalone, inspecter dépendances
Fonctionne seulement depuis le repo	Chemin relatif vers source	Utiliser chemins runtime robustes
Onefile KO, standalone OK	Extraction temporaire / chemin	Corriger accès fichiers, rester standalone si besoin

---

Rapport final de validation

Pour un livrable sérieux, il faut produire un petit rapport de validation. Cela permet de savoir exactement ce qui a été testé et d'éviter les builds impossibles à reproduire.

Structure recommandée

```
build/reports/
                                        │
                                        ├── compilation-report.xml
                                        ├── validation-report.md
                                        ├── environment.txt
                                        ├── command.txt
                                        ├── source-output.txt
                                        ├── binary-output.txt
                                        ├── pytest-output.txt
                                        └── performance-baseline.txt
```

Contenu minimum du rapport

Section	Contenu
Environnement	OS, Python, Nuitka, compilateur
Commande build	Commande exacte utilisée
Tests source	Résultat et code retour
Tests binaires	Résultat et code retour
Packaging	Standalone / onefile / machine cible
Performance	Temps moyen, mémoire si disponible
Décision	GO, NO-GO ou GO avec réserve

Template Markdown de rapport

```
# Nuitka Validation Report

                        ## Environment
                        - OS:
                        - Python:
                        - Nuitka:
                        - Compiler:
                        - Build date:

                        ## Build command
                        python -m nuitka ...

                        ## Source validation
                        - Command:
                        - Exit code:
                        - Result:

                        ## Binary validation
                        - Artifact:
                        - Command:
                        - Exit code:
                        - Result:

                        ## Packaging validation
                        - Standalone tested: yes/no
                        - Onefile tested: yes/no
                        - Clean machine tested: yes/no
                        - Missing files: none/list

                        ## Performance baseline
                        - Source average:
                        - Binary average:
                        - Notes:

                        ## Decision
                        - Status: GO / NO-GO / GO WITH RESERVES
                        - Comments:
```

Conclusion : la validation doit devenir un rituel : source OK, tests OK, build OK, binaire OK, packaging OK, rapport OK. Sans cela, le build n'est pas industrialisable.

4.2 Cas réel : compiler le cœur d'un addon Django — ideolab_admin_tools

Objectif : protéger la valeur métier sans casser l'intégration Django

L'objectif n'est pas de compiler tout Django, ni de transformer l'addon en bloc opaque impossible à maintenir. L'approche raisonnable consiste à garder l'intégration Django lisible — `apps.py`, `admin.py`, `views.py`, `urls.py`, templates, static — et à compiler uniquement le cœur propriétaire : analyse, introspection, scoring, génération de métadonnées, règles métier.

Approche recommandée

```
Django addon
                                    │
                                    ├── Couche Django publique
                                    │   ├── apps.py
                                    │   ├── admin.py
                                    │   ├── views.py
                                    │   ├── urls.py
                                    │   ├── templates/
                                    │   └── static/
                                    │
                                    ├── API stable Python
                                    │   └── core_api.py
                                    │
                                    └── Cœur propriétaire compilé
                                    ├── protected_core/scoring.pyd / .so
                                    ├── protected_core/analyzer.pyd / .so
                                    ├── protected_core/rules_engine.pyd / .so
                                    └── protected_core/metadata_builder.pyd / .so
```

> **Principe clé :** Django reste Django. Le cœur métier devient binaire. Cela protège mieux le savoir-faire sans rendre l'addon ingérable.

Pourquoi ne pas compiler tout l'addon ?

Élément	Compiler ?	Raison
`apps.py`	Non	Glue code Django, peu sensible.
`admin.py`	Non au début	Introspection Django, registre admin, debug plus simple.
`views.py`	Non au début	Couche HTTP, templates, permissions.
`templates/`	Non	HTML à livrer comme fichiers data.
`static/`	Non	CSS / JS / assets.
`protected_core/`	Oui	Valeur métier propriétaire.

But business

Pour un addon comme `ideolab_admin_tools`, la valeur n'est pas seulement dans l'affichage des pages Django Admin. Elle est dans la logique qui analyse les modèles, détecte les patterns, calcule des scores, construit des diagnostics, génère des métadonnées, ou produit des recommandations. C'est cette logique qu'il faut isoler puis compiler.

Définir le périmètre compilable

Avant Nuitka, il faut choisir un périmètre stable, testable et suffisamment indépendant. Le pire choix serait de compiler un fichier qui dépend directement de tout Django, du request object, des templates, de la session, de l'admin, des settings et de la base.

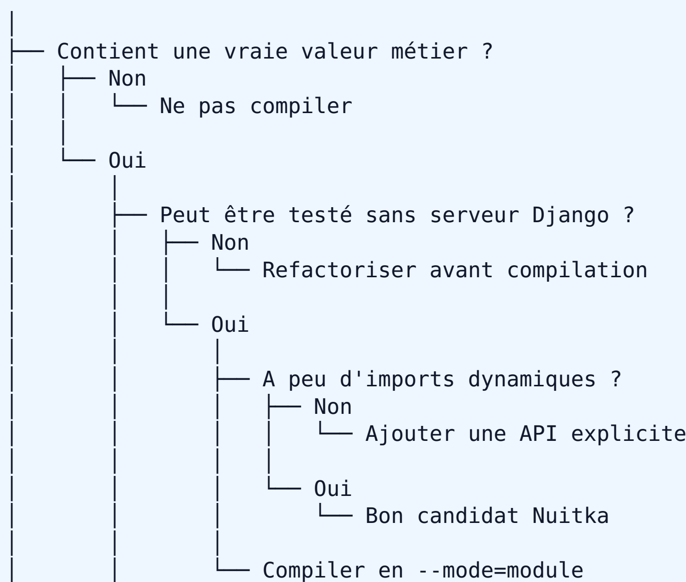
Bon périmètre

Module	Pourquoi c'est un bon candidat
`scoring.py`	Calcul métier isolable et testable.
`rules_engine.py`	Contient des règles propriétaires.
`analyzer.py`	Algorithmes de diagnostic et heuristiques.
`metadata_builder.py`	Transformation structurée des infos Django.
`inspector_core.py`	Peut recevoir des données normalisées en entrée.

Mauvais périmètre au début

Fichier	Pourquoi éviter
`admin.py`	Fortement couplé au registre Django Admin.
`views.py`	Dépend de request, permissions, templates.
`models.py`	Lié à l'ORM, migrations, app registry.
`urls.py`	Aucune valeur métier sensible.
`migrations/`	Doit rester lisible et traçable.

Arbre de décision



```
Module candidat
                        │
                        ├── Contient une vraie valeur métier ?
                        │   ├── Non
                        │   │   └── Ne pas compiler
                        │   │
                        │   └── Oui
                        │       │
                        │       ├── Peut être testé sans serveur Django ?
                        │       │   ├── Non
                        │       │   │   └── Refactoriser avant compilation
                        │       │   │
                        │       │   └── Oui
                        │       │       │
                        │       │       ├── A peu d'imports dynamiques ?
                        │       │       │   ├── Non
                        │       │       │   │   └── Ajouter une API explicite
                        │       │       │   │
                        │       │       │   └── Oui
                        │       │       │       └── Bon candidat Nuitka
                        │       │       │
                        │       │       └── Compiler en --mode=module
```

Objectif de conception : le module compilé doit recevoir des données simples, retourner des structures simples, et éviter de manipuler directement l'objet request, le registre admin ou les templates.

Structure cible de l'addon

La structure doit séparer clairement la couche Django, l'API publique, le cœur protégé et les éventuels fichiers de fallback utilisés uniquement en développement.

Arborescence recommandée

```
ideolab_admin_tools/
                                │
                                ├── __init__.py
                                ├── apps.py
                                ├── admin.py
                                ├── urls.py
                                ├── views.py
                                │
                                ├── templates/
                                │   └── ideolab_admin_tools/
                                │       ├── dashboard.html
                                │       └── modal_detail.html
                                │
                                ├── static/
                                │   └── ideolab_admin_tools/
                                │       ├── css/
                                │       └── js/
                                │
                                ├── core_api.py
                                ├── normalizers.py
                                ├── dto.py
                                │
                                └── protected_core/
                                ├── __init__.py
                                ├── scoring.py             # source dev
                                ├── scoring_py.py          # fallback dev optionnel
                                ├── analyzer.py            # source dev
                                ├── analyzer_py.py         # fallback dev optionnel
                                ├── rules_engine.py        # source dev
                                ├── metadata_builder.py    # source dev
                                └── rules/
                                └── default_rules.json
```

Après compilation

```
ideolab_admin_tools/protected_core/
                                │
                                ├── __init__.py
                                │
                                ├── scoring.cp312-win_amd64.pyd
                                ├── analyzer.cp312-win_amd64.pyd
                                ├── rules_engine.cp312-win_amd64.pyd
                                ├── metadata_builder.cp312-win_amd64.pyd
                                │
                                ├── scoring_py.py        # seulement si fallback dev autorisé
                                ├── analyzer_py.py       # seulement si fallback dev autorisé
                                │
                                └── rules/
                                └── default_rules.json
```

Attention : les fichiers `.pyd` Windows et `.so` Linux sont spécifiques à l'OS, à l'architecture et souvent à la version Python.

Séparation des responsabilités

Couche	Rôle	Compilation
Django integration	Admin, URLs, views, permissions, templates.	Non au début
API publique	Façade stable appelée par Django.	Non, garder lisible
DTO / normalizers	Transformer objets Django en dict/list simples.	Optionnel
Protected core	Scoring, règles, analyse, heuristiques.	Oui
Data rules	JSON/YAML de règles.	Non, inclure comme data

Créer une API Python stable autour du module compilé

Les vues Django, l'admin et les templates ne doivent pas appeler directement les détails internes du module compilé. Ils doivent passer par une façade Python stable. Cette façade permet de gérer les erreurs, le fallback de développement, les logs, les conversions de données et les futures évolutions.

Exemple `core_api.py`

```
from __future__ import annotations

                                import logging
                                import os
                                from typing import Any

                                logger = logging.getLogger(__name__)

                                ALLOW_PY_FALLBACK =
os.environ.get("IDEOLAB_ADMIN_TOOLS_ALLOW_PY_FALLBACK") == "1"


                                def _load_scoring_engine():
                                try:
                                from .protected_core.scoring import compute_admin_score
                                return compute_admin_score
                                except Exception as exc:
                                logger.exception("Unable to import compiled scoring engine")

                                if not ALLOW_PY_FALLBACK:
                                raise RuntimeError(
                                "Compiled scoring engine is unavailable and Python fallback is disabled."
                                ) from exc

                                from .protected_core.scoring_py import compute_admin_score
                                logger.warning("Using Python fallback scoring engine")
                                return compute_admin_score


                                def score_model_admin(model_admin_class: type[Any]) -> dict[str, Any]:
                                """
                                Public stable API used by Django admin/views.

                                The Django layer should call this function and not the compiled module
directly.
                                """
                                compute_admin_score = _load_scoring_engine()
                                return compute_admin_score(model_admin_class)
```

Pourquoi cette façade est importante ?

Besoin	Rôle de la façade
Stabilité	Les appels Django ne changent pas si le moteur interne change.
Fallback dev	Permet de travailler sans recompiler à chaque modification.
Logs	Centralise les erreurs d'import du module compilé.
Contrôle prod	Interdit le fallback Python en production.
Tests	Permet de tester l'API publique plutôt que les détails internes.

Bon pattern : l'admin Django appelle `score_model_admin()`. Il ne sait pas si derrière c'est du Python pur, un `.pyd` ou un `.so`.

Exemple côté admin Django

```
# ideolab_admin_tools/admin.py

                        from __future__ import annotations

                        from django.contrib import admin
                        from django.http import JsonResponse
                        from django.urls import path

                        from .core_api import score_model_admin


                        class IdeolabAdminToolsSiteMixin:
                        def get_urls(self):
                        urls = super().get_urls()
                        custom_urls = [
                        path(
                        "ideolab-admin-tools/score/",
                        self.admin_view(self.ideolab_score_view),
                        name="ideolab_admin_tools_score",
                        ),
                        ]
                        return custom_urls + urls

                        def ideolab_score_view(self, request):
                        result = score_model_admin(type(self))
                        return JsonResponse(result)
```

Compiler un module d'addon Django avec Nuitka

La première compilation doit viser un seul module simple. Le bon candidat typique est `protected_core/scoring.py`. Une fois ce premier module validé, on peut élargir vers `analyzer.py`, `rules_engine.py`, puis éventuellement un sous-package.

Préparer l'environnement

```
cd path/to/project

                                python -m venv .venv

                                # Windows
                                .venv\Scripts\activate.bat

                                # Linux / macOS
                                source .venv/bin/activate

                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -U nuitka ordered-set zstandard
```

Compiler `scoring.py`

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                --report=build/nuitka_modules/scoring-report.xml \
                                ideolab_admin_tools/protected_core/scoring.py
```

Résultats attendus

```
build/nuitka_modules/
                                │
                                ├── scoring.build/
                                │   └── fichiers intermédiaires C/C++
                                │
                                └── scoring.cp312-win_amd64.pyd
                                ou
                                └── scoring.cpython-312-x86_64-linux-gnu.so
```

Copie dans l'addon

```
# Exemple Windows
                                copy build\nuitka_modules\scoring*.pyd ideolab_admin_tools\protected_core\

                                # Exemple Linux / macOS
                                cp build/nuitka_modules/scoring*.so ideolab_admin_tools/protected_core/
```

Important : le nom généré dépend de la version Python et de l'OS. Ne pas renommer à la main sans comprendre l'ABI.

Build script conseillé

```
# scripts/build_nuitka_scoring.sh
                        #!/usr/bin/env bash
                        set -euo pipefail

                        mkdir -p build/nuitka_modules build/logs build/reports

                        python -m nuitka \
                        --mode=module \
                        --output-dir=build/nuitka_modules \
                        --remove-output \
                        --show-progress \
                        --report=build/reports/scoring-report.xml \
                        ideolab_admin_tools/protected_core/scoring.py \
                        2>&1 | tee build/logs/scoring-build.log

                        cp build/nuitka_modules/scoring*.so ideolab_admin_tools/protected_core/

                        python -c "import ideolab_admin_tools.protected_core.scoring; print('compiled
import OK')"
```

Packaging Python : inclure les extensions compilées

Un addon Django doit rester installable via pip, wheel, archive ou dépôt Git. Si des modules compilés sont ajoutés, il faut s'assurer qu'ils sont inclus dans le package et que les templates/static restent disponibles.

`MANIFEST.in`

```
recursive-include ideolab_admin_tools/templates *
                                recursive-include ideolab_admin_tools/static *
                                recursive-include ideolab_admin_tools/protected_core *.pyd
                                recursive-include ideolab_admin_tools/protected_core *.so
                                recursive-include ideolab_admin_tools/protected_core *.dll
                                recursive-include ideolab_admin_tools/protected_core/rules *.json
                                recursive-exclude ideolab_admin_tools/protected_core *_dev.py
                                recursive-exclude ideolab_admin_tools/protected_core *_scratch.py
                                recursive-exclude tests *
                                recursive-exclude docs/internal *
```

`pyproject.toml` indicatif

```
[build-system]
                                requires = ["setuptools", "wheel"]
                                build-backend = "setuptools.build_meta"

                                [tool.setuptools]
                                include-package-data = true

                                [tool.setuptools.packages.find]
                                include = ["ideolab_admin_tools*"]
```

Ce qu'il faut livrer

Élément	Inclure ?	Pourquoi
`templates/`	Oui	Rendu Django Admin.
`static/`	Oui	CSS / JS / images.
`*.pyd`	Oui Windows	Extension compilée Windows.
`*.so`	Oui Linux/macOS	Extension compilée Unix.

Élément	Inclure ?	Pourquoi
`rules/*.json`	Oui si utilisé	Règles runtime.
`.env`	Non	Secrets.
`tests/`	Non pour release fermée	Révèle trop de logique interne.

Attention : compiler ne protège pas si le package final embarque aussi les sources Python sensibles à côté des extensions compilées.

Construire une wheel

```
python -m pip install build

                        python -m build

                        # Résultat attendu :
                        # dist/ideolab_admin_tools-0.1.0-py3-none-any.whl
                        # ou une wheel spécifique plateforme si packaging binaire avancé
```

Settings, fallback et comportement dev/prod

Le fallback Python est très utile en développement, mais dangereux en production : il peut donner l'impression que le module compilé fonctionne alors que l'application utilise en réalité le source Python. Il faut donc contrôler explicitement ce comportement.

Règle recommandée

Environnement	Fallback Python	Comportement
Dev local	Autorisé	Permet d'itérer sans recompiler à chaque modification.
CI build	Désactivé après build	Force le test du module compilé.
Staging	Désactivé	Identique production.
Production	Interdit	Erreur explicite si extension absente.

Variables d'environnement

```
# Dev seulement
                                IDEOLAB_ADMIN_TOOLS_ALLOW_PY_FALLBACK=1

                                # Staging / Production
                                IDEOLAB_ADMIN_TOOLS_ALLOW_PY_FALLBACK=0

                                # Optionnel : vérifier mode strict
                                IDEOLAB_ADMIN_TOOLS_REQUIRE_COMPILED_CORE=1
```

Conseil : afficher dans les logs au démarrage si le moteur utilisé est `compiled` ou `python fallback`.

Check de démarrage Django

```
# ideolab_admin_tools/apps.py

                        from __future__ import annotations

                        import logging
                        from django.apps import AppConfig

                        logger = logging.getLogger(__name__)


                        class IdeolabAdminToolsConfig(AppConfig):
                        default_auto_field = "django.db.models.BigAutoField"
                        name = "ideolab_admin_tools"
                        verbose_name = "IDEO-Lab Admin Tools"

                        def ready(self):
                        try:
                        from .protected_core import scoring
                        logger.info("IDEO-Lab Admin Tools: compiled scoring engine loaded: %s", scoring)
                        except Exception:
                        logger.exception("IDEO-Lab Admin Tools: compiled scoring engine is not available")
```

Tests Django à exécuter

Une fois le module compilé copié dans l'addon, il faut tester à trois niveaux : import Python, checks Django, puis navigation réelle dans l'admin.

Commandes de test

```
# Vérifier l'import direct du module compilé
                                python -c "import ideolab_admin_tools.protected_core.scoring;
print('import OK')"

                                # Vérifier l'API publique
                                python -c "from ideolab_admin_tools.core_api import score_model_admin;
print(score_model_admin)"

                                # Checks Django
                                python manage.py check

                                # Tests de l'app
                                python manage.py test ideolab_admin_tools

                                # Lancer le serveur
                                python manage.py runserver
```

Tests manuels Admin

Action	Résultat attendu
Ouvrir `/admin/`	Admin charge sans erreur.
Ouvrir les pages custom admin tools	Pages visibles.
Cliquer sur boutons / modales	JS et endpoints OK.
Appeler endpoint JSON scoring	Réponse JSON cohérente.
Lire logs serveur	Aucun `ImportError`.

Risques et tests associés

Risque	Symptôme	Test	Correction
ImportError du module compilé	Erreur au démarrage	`python -c "import ...scoring"`	Recompiler pour le bon OS/ Python.
Extension incompatible	Erreur ABI / DLL	Tester sur Python cible	Builder par version Python.
Fallback dev accidentel	Prod utilise `scoring_py.py`	Variable fallback à 0	Interdire fallback prod.
Data rules absentes	`FileNotFoundError`	Endpoint qui lit les règles	Inclure `rules/*.json`.
Template absent	TemplateDoesNotExist	Ouvrir page admin	Corriger package data.

> **Validation minimale :** import direct OK, `manage.py check` OK, tests OK, admin accessible, endpoint scoring OK, aucun fallback Python en prod.

Release : produire un livrable propre

Le release process doit éviter deux erreurs : livrer les sources sensibles par accident, ou livrer une extension compilée incompatible avec l'environnement cible.

Pipeline de release

```
Release pipeline
│
├── 1. Clean workspace
├── 2. Run source tests
├── 3. Build compiled modules
├── 4. Copy .pyd / .so into package
├── 5. Remove sensitive .py sources if needed
├── 6. Build wheel / archive
├── 7. Install into clean venv
├── 8. Run Django checks
├── 9. Run admin smoke tests
└── 10. Tag release
```

Checklist release

#	Contrôle	Statut attendu
1	Tests source	OK
2	Build Nuitka	OK
3	Import module compilé	OK
4	Fallback prod	Désactivé
5	Templates/static inclus	OK
6	Sources sensibles absentes	OK selon stratégie
7	Wheel installée dans venv propre	OK
8	Django admin smoke test	OK

Test dans un venv propre

```
python -m venv /tmp/ideolab_admin_tools_release_test
                        source /tmp/ideolab_admin_tools_release_test/bin/activate

                        python -m pip install --upgrade pip
                        python -m pip install dist/ideolab_admin_tools-*.whl

                        python -c "import ideolab_admin_tools; print('addon import OK')"
                        python -c "import ideolab_admin_tools.protected_core.scoring; print('compiled core
OK')"
```

> **Point critique :** si la wheel contient à la fois `scoring.py` sensible et `scoring.pyd`, la protection est fortement diminuée. Il faut décider clairement ce qui reste source et ce qui devient uniquement binaire.

Roadmap progressive pour ideolab_admin_tools

La bonne stratégie consiste à avancer progressivement. On commence par un module très petit, puis on élargit quand les tests, le packaging et les imports sont maîtrisés.

Étape	Objectif	Action	Critère GO
R1	Identifier le cœur métier	Lister scoring, analyzer, rules engine.	Liste validée.
R2	Créer une API stable	Ajouter `core_api.py`.	Django appelle l'API, pas le core directement.
R3	Compiler un module simple	`scoring.py` en `--mode=module`.	Import direct OK.
R4	Tester dans Django	`manage.py check`, admin, endpoint.	Aucune erreur runtime.
R5	Packaging	Inclure `.pyd` / `.so`, templates, static.	Wheel installable.
R6	Désactiver fallback prod	Variable stricte.	Prod échoue si core compilé absent.
R7	Compiler plusieurs modules	analyzer, rules_engine, metadata_builder.	Tests verts.
R8	Release multi-plateforme	Build Windows/Linux par version Python.	Artefacts séparés et documentés.

Premier module recommandé

```
# ideolab_admin_tools/protected_core/scoring.py

                                from __future__ import annotations
                                from typing import Any


                                def compute_admin_score(model_admin_class: type[Any]) -> dict[str, Any]:
                                score = 0
                                signals = []

                                if hasattr(model_admin_class, "list_display"):
                                score += 10
                                signals.append("list_display")

                                if hasattr(model_admin_class, "search_fields"):
                                score += 10
                                signals.append("search_fields")

                                if hasattr(model_admin_class, "list_filter"):
                                score += 10
                                signals.append("list_filter")

                                return {
                                "score": score,
                                "signals": signals,
                                "level": "advanced" if score >= 25 else "basic",
                                }
```

Commande de premier essai

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                ideolab_admin_tools/protected_core/scoring.py

                                python -c "import ideolab_admin_tools.protected_core.scoring; print('OK')"

                                python manage.py check
```

Conclusion : pour `ideolab_admin_tools`, le meilleur premier objectif est un module `scoring` compilé, importé par une API Python stable, testé dans Django Admin, puis packagé proprement.

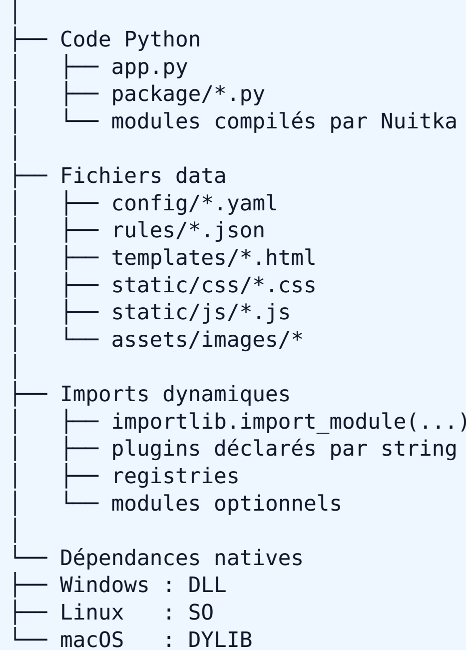
5.1 Fichiers, templates, assets, DLL — inclure les ressources runtime avec Nuitka

Pourquoi les fichiers externes sont critiques ?

Nuitka compile le code Python, mais une application réelle ne se limite jamais au code. Elle dépend souvent de fichiers JSON, YAML, templates HTML, fichiers CSS/JS, images, certificats, fichiers SQL, règles métier, dictionnaires, modèles ML, DLL, librairies natives et modules chargés dynamiquement. Si ces ressources ne sont pas incluses ou localisées correctement, le binaire compilé peut fonctionner en développement puis échouer immédiatement en production.

Carte mentale des ressources runtime

Application Python compilée



Idée centrale : un build Nuitka réussi doit embarquer le code, mais aussi tout ce que le code lit, importe ou charge au runtime.

Symptômes typiques si une ressource manque

Symptôme	Cause probable	Famille
`FileNotFoundError`	Fichier JSON/YAML absent	Data file
`TemplateDoesNotExist`	Dossier templates absent	Templates
CSS / JS absent	Dossier static non livré	Assets
`ModuleNotFoundError`	Import dynamique non détecté	Imports
`DLL load failed`	Librairie native absente	Native dependency
Fonctionne depuis le repo mais pas ailleurs	Chemin relatif vers les sources	Runtime path

Règle professionnelle : toujours tester le binaire hors du dossier source. Sinon, le programme peut lire par accident les fichiers du repo et masquer un packaging cassé.

Types de ressources à identifier avant compilation

Avant d'écrire une commande Nuitka, il faut faire l'inventaire des ressources consommées par l'application. Cela évite les builds qui fonctionnent seulement sur la machine du développeur.

Type	Exemples	Risque si absent	Option Nuitka typique
Config	`config/defaults.yaml`, `.toml`, `.ini`	Application non configurable	`--include-data-files`
Règles métier	`rules/*.json`, `rules/*.yaml`	Moteur de règles inutilisable	`--include-data-dir`
Templates	`templates/*.html`	Erreur template introuvable	`--include-data-dir`
Static	`static/css`, `static/js`, images	Interface cassée	`--include-data-dir`
Certificats	`certs/ca.pem`	Erreur TLS / SSL	`--include-data-files`
SQL	`sql/schema.sql`, `fixtures/*.sql`	Initialisation impossible	`--include-data-dir`
Plugins Python	`myapp.plugins.csv_exporter`	ImportError au runtime	`--include-module`
Packages dynamiques	`myapp.rules`, `myapp.backends`	Modules non inclus	`--include-package`
Native libs	`.dll`, `.so`, `.dylib`	Erreur de chargement native	Standalone + inspection `.dist`

Inventaire recommandé

```
project/
                                        │
                                        ├── config/
                                        │   └── defaults.yaml
                                        │
                                        ├── rules/
                                        │   ├── default_rules.json
                                        │   └── admin_scoring.json
                                        │
                                        ├── templates/
                                        │   └── report.html
                                        │
                                        ├── static/
                                        │   ├── css/app.css
                                        │   └── js/app.js
                                        │
                                        ├── certs/
                                        │   └── ca.pem
                                        │
                                        └── plugins/
                                        ├── csv_exporter.py
                                        └── html_exporter.py
```

À ne jamais inclure par erreur

Élément	Pourquoi l'exclure
`.env`	Contient souvent des secrets.
`.git/`	Historique complet du code source.
`tests/`	Révèle les comportements internes attendus.
`docs/internal/`	Documentation privée d'architecture.
`dev_config.yaml`	Chemins et paramètres de développement.

> **Important :** un binaire compilé ne protège rien si le livrable embarque les sources sensibles, les secrets ou les fichiers internes à côté.

Commandes Nuitka pour inclure les ressources

Les options principales sont `--include-data-files` pour des fichiers précis, `--include-data-dir` pour des dossiers, `--include-module` pour un module dynamique, et `--include-package` pour un package complet.

Inclure des fichiers et dossiers

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --include-data-files=config/defaults.yaml=config/defaults.yaml \
                                --include-data-files=rules/*.json=rules/ \
                                --include-data-dir=templates=templates \
                                --include-data-dir=static=static \
                                app.py
```

Inclure des certificats

```
python -m nuitka \
                                --mode=standalone \
                                --include-data-files=certs/ca.pem=certs/ca.pem \
                                app.py
```

Inclure imports dynamiques

```
python -m nuitka \
                                --mode=standalone \
                                --include-module=my_package.dynamic_plugin \
                                --include-module=my_package.exporters.csv_exporter \
                                --include-package=my_package.rules \
                                app.py
```

Build avec rapport

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --report=build/reports/data-report.xml \
                                --show-progress \
                                --include-data-dir=templates=templates \
                                --include-data-dir=static=static \
                                app.py
```

Différence entre fichier et dossier

Option	Usage	Exemple	Résultat
`--include-data-files`	Inclure un fichier ou un pattern	`config/default.yaml=config/default.yaml`	Le fichier est copié à l'emplacement cible.
`--include-data-dir`	Inclure un dossier complet	`templates=templates`	Le dossier entier est copié.

Option	Usage	Exemple	Résultat
`--include-module`	Inclure un module Python précis	`myapp.plugins.pdf`	Le module est intégré au graphe d'import.
`--include-package`	Inclure un package Python complet	`myapp.plugins`	Tout le package est considéré.

Conseil : au début, inclure explicitement les ressources nécessaires. Une commande longue mais claire est préférable à un build mystérieux qui fonctionne par hasard.

Chemins runtime : le piège numéro 1

Le code source lit souvent des fichiers avec des chemins relatifs au dossier du projet. Après compilation, ce dossier n'existe plus forcément. En mode standalone, les fichiers sont dans le dossier `.dist`. En mode onefile, ils peuvent être extraits dans un dossier temporaire.

Mauvais pattern

```
from pathlib import Path

                                    # Fragile : dépend du dossier depuis lequel on lance le programme
                                    rules_path = Path("rules/default_rules.json")
                                    content = rules_path.read_text(encoding="utf-8")
```

Pourquoi c'est fragile ?

- Le working directory peut changer.
- Le binaire peut être lancé depuis un autre dossier.
- Le mode onefile peut extraire les fichiers ailleurs.
- Le dossier source peut ne pas exister chez l'utilisateur final.

Meilleur pattern

```
from pathlib import Path
                                    import sys


                                    def runtime_base_dir() -> Path:
                                    """
                                    Return a robust base directory for source, standalone and onefile
contexts.
                                    """
                                    if getattr(sys, "frozen", False):
                                    return Path(sys.executable).resolve().parent

                                    return Path(__file__).resolve().parent


                                    def load_default_rules() -> str:
                                    base_dir = runtime_base_dir()
                                    rules_path = base_dir / "rules" / "default_rules.json"
                                    return rules_path.read_text(encoding="utf-8")
```

But : toujours construire les chemins à partir d'un point de référence clair, et non à partir du dossier courant implicite.

Tester les chemins hors dossier source

```
# Linux / macOS
                        mkdir -p /tmp/nuitka_path_test
                        cp -r build/nuitka/app.dist /tmp/nuitka_path_test/
                        cd /tmp/nuitka_path_test/app.dist
                        ./app

                        # Windows PowerShell
                        New-Item -ItemType Directory -Force C:\temp\nuitka_path_test
                        Copy-Item -Recurse build\nuitka\app.dist C:\temp\nuitka_path_test\
                        cd C:\temp\nuitka_path_test\app.dist
                        .\app.exe
```

Validation réelle : si le programme fonctionne seulement depuis le repo, le packaging n'est pas validé.

Django : templates, static, fichiers package

Dans un addon Django, les templates et fichiers static doivent rester des fichiers livrés avec le package. Nuitka n'a pas vocation à compiler le HTML, le CSS ou le JavaScript. Il faut les inclure proprement dans le package Python, puis les rendre disponibles au runtime.

Arborescence addon Django

```
ideolab_admin_tools/
                                │
                                ├── apps.py
                                ├── admin.py
                                ├── views.py
                                ├── urls.py
                                │
                                ├── templates/
                                │   └── ideolab_admin_tools/
                                │       ├── dashboard.html
                                │       ├── modal_detail.html
                                │       └── components/
                                │
                                ├── static/
                                │   └── ideolab_admin_tools/
                                │       ├── css/admin_tools.css
                                │       ├── js/admin_tools.js
                                │       └── img/logo.svg
                                │
                                └── protected_core/
                                ├── scoring.pyd / scoring.so
                                └── rules/
                                └── default_rules.json
```

Packaging Django classique

```
# MANIFEST.in
                                recursive-include ideolab_admin_tools/templates *
                                recursive-include ideolab_admin_tools/static *
                                recursive-include ideolab_admin_tools/protected_core *.pyd
                                recursive-include ideolab_admin_tools/protected_core *.so
                                recursive-include ideolab_admin_tools/protected_core/rules *.json
                                recursive-exclude tests *
                                recursive-exclude docs/internal *
```

pyproject.toml

```
[tool.setuptools]
                                include-package-data = true

                                [tool.setuptools.packages.find]
                                include = ["ideolab_admin_tools*"]
```

Commandes de test Django

```
# Vérifier que Django voit l'addon
                        python manage.py check

                        # Vérifier les templates
                        python manage.py shell -c "from django.template.loader import get_template;
print(get_template('ideolab_admin_tools/dashboard.html'))"

                        # Vérifier les static en dev
                        python manage.py findstatic ideolab_admin_tools/css/admin_tools.css

                        # Vérifier l'import du cœur compilé
                        python -c "import ideolab_admin_tools.protected_core.scoring; print('compiled core
OK')"
```

Pour un addon Django : templates/static relèvent du packaging Python et Django, tandis que le cœur métier peut être compilé avec Nuitka en module `.pyd` ou `.so`.

Imports dynamiques : le piège invisible

Nuitka suit naturellement les imports explicites. En revanche, si le code construit un nom de module sous forme de chaîne de caractères puis l'importe avec `importlib`, Nuitka ne peut pas toujours deviner les modules à inclure.

Exemple fragile

```
import importlib

                                PLUGIN_NAME = "myapp.plugins.csv_exporter"

                                plugin_module = importlib.import_module(PLUGIN_NAME)
                                plugin_module.run()
```

Ici, le module est chargé par son nom texte. Dans un système réel, ce nom peut venir d'un fichier YAML, d'un setting Django, d'une base de données ou d'un registre de plugins.

Commande de correction

```
python -m nuitka \
                                --mode=standalone \
                                --include-module=myapp.plugins.csv_exporter \
                                --include-module=myapp.plugins.html_exporter \
                                --include-package=myapp.rules \
                                app.py
```

Règle : tout module chargé par nom texte doit être listé explicitement, ou inclus via son package parent.

Table de diagnostic imports dynamiques

Pattern dans le code	Risque Nuitka	Action
`importlib.import_module(name)`	Module non détecté	`--include-module`
`__import__(name)`	Module non détecté	`--include-module`
Plugin déclaré dans YAML	Module invisible à l'analyse statique	`--include-package`
Django setting avec dotted path	Classe ou backend absent	Inclure le module backend
Entry points / extensions	Découverte runtime incomplète	Plugin Nuitka ou include explicite

Test d'import dynamique

```
python - <<'PY'
                        import importlib

                        modules = [
                        "myapp.plugins.csv_exporter",
                        "myapp.plugins.html_exporter",
                        "myapp.rules.default_rules",
                        ]

                        for module_name in modules:
                        module = importlib.import_module(module_name)
                        print("OK:", module_name, module)
                        PY
```

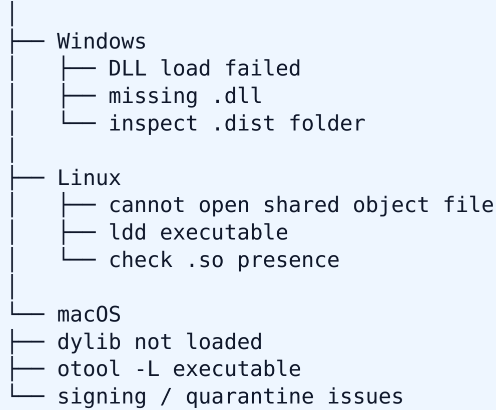
DLL, SO, DYLIB : dépendances natives

Certaines librairies Python s'appuient sur du code natif. Sous Windows, ce sont souvent des fichiers `.dll`. Sous Linux, des `.so`. Sous macOS, des `.dylib`. En mode standalone, Nuitka tente d'inclure les dépendances nécessaires, mais il faut toujours vérifier.

Dépendances natives typiques

Type de projet	Exemples de dépendances natives
Base de données	Clients PostgreSQL, MySQL, MariaDB, SQLite natif.
Cryptographie	OpenSSL, libs crypto.
Compression	zlib, zstandard, brotli.
XML / HTML parsing	libxml2, libxslt.
Scientific stack	numpy, scipy, pandas, BLAS.
GUI	Qt, Tk, GTK.

Diagnostic natif



```
Runtime native error
                                │
                                ├── Windows
                                │   ├── DLL load failed
                                │   ├── missing .dll
                                │   └── inspect .dist folder
                                │
                                ├── Linux
                                │   ├── cannot open shared object file
                                │   ├── ldd executable
                                │   └── check .so presence
                                │
                                └── macOS
                                ├── dylib not loaded
                                ├── otool -L executable
                                └── signing / quarantine issues
```

> **Conseil :** toujours valider le standalone sur une machine aussi proche que possible de la machine cible.

Commandes d'inspection

Linux

```
# Voir les dépendances natives
                                ldd build/nuitka/app.dist/app

                                # Chercher les .so livrées
                                find build/nuitka/app.dist -name "*.so*" -type f
```

Windows PowerShell

```
# Lister les DLL dans le standalone
                                Get-ChildItem -Recurse build\nuitka\app.dist -Filter *.dll

                                # Vérifier la présence de l'exécutable
                                Get-ChildItem build\nuitka\app.dist
```

Debug : relier symptôme, cause et correction

Les erreurs liées aux ressources sont très fréquentes. Le diagnostic doit partir du message d'erreur, puis remonter vers le type de ressource manquante : fichier data, template, static, module dynamique ou dépendance native.

Symptôme	Cause probable	Solution Nuitka / packaging	Test de validation
`FileNotFoundError`	Fichier non inclus ou chemin fragile	`--include-data-files` + chemin runtime robuste	Lancer hors dossier source
`TemplateDoesNotExist`	Dossier templates absent du livrable	`--include-data-dir=templates=templates` ou packaging Django	`get_template(...)`
CSS / JS absent	Static non inclus ou non collecté	Inclure static ou corriger packaging Django	Ouvrir l'UI finale
`ModuleNotFoundError`	Import dynamique absent	`--include-module` ou `--include-package`	Importer explicitement le module

Symptôme	Cause probable	Solution Nuitka / packaging	Test de validation
`DLL load failed`	DLL native absente ou incompatible	Inspecter `.dist`, recompiler sur bonne plateforme	Tester sur machine cible
Standalone OK, onefile KO	Chemins temporaires onefile	Corriger accès fichiers ou rester en standalone	Comparer standalone vs onefile

---

Script de diagnostic ressources

```
from __future__ import annotations

                        import importlib
                        from pathlib import Path


                        def check_file(path: str) -> None:
                        p = Path(path)
                        if not p.exists():
                        raise FileNotFoundError(f"Missing resource: {p}")
                        print("OK file:", p)


                        def check_module(module_name: str) -> None:
                        module = importlib.import_module(module_name)
                        print("OK module:", module_name, module)


                        def main() -> int:
                        check_file("config/defaults.yaml")
                        check_file("rules/default_rules.json")
                        check_module("my_package.dynamic_plugin")
                        print("OK - resource diagnostic passed")
                        return 0


                        if __name__ == "__main__":
                        raise SystemExit(main())
```

Bonne pratique : créer un mode `--self-check` dans vos CLI compilés pour vérifier les fichiers, imports et dépendances critiques au démarrage.

Packaging release : livrer uniquement ce qui est nécessaire

Un livrable propre contient les ressources nécessaires à l'exécution, mais pas les secrets, fichiers de développement, tests internes ou sources sensibles. Pour un addon Django, il faut aussi vérifier que `MANIFEST.in` et le packaging Python incluent bien templates, static et extensions binaires.

Livrable standalone

```
release/
                                         │
                                         ├── app.dist/
                                         │   ├── app.exe / app
                                         │   ├── config/
                                         │   │   └── defaults.yaml
                                         │   ├── rules/
                                         │   │   └── default_rules.json
                                         │   ├── templates/
                                         │   ├── static/
                                         │   ├── *.dll / *.so / *.dylib
                                         │   └── runtime files
                                         │
                                         └── README_RUN.txt
```

Livrable addon Django

```
wheel contents
                                         │
                                         ├── ideolab_admin_tools/
                                         │   ├── apps.py
                                         │   ├── admin.py
                                         │   ├── views.py
                                         │   ├── templates/
                                         │   ├── static/
                                         │   └── protected_core/
                                         │       ├── scoring.pyd / scoring.so
                                         │       └── rules/default_rules.json
                                         │
                                         └── metadata
```

Table de nettoyage release

À exclure	Raison	Méthode
`.env`	Secrets	Ne jamais inclure
`.git/`	Historique source	Exclure du package
`tests/`	Logique interne	Exclure release fermée
`*.py` sensibles	Sources propriétaires	Remplacer par `.pyd` / `.so` si stratégie fermée
`dev_config.*`	Config locale	Remplacer par config publique
`build intermediates`	Pollution / volume	Ne livrer que dist utile

Point critique : le packaging est une surface de fuite. Toujours inspecter le contenu final avant livraison.

Checklist finale : ressources, imports, DLL, templates

#	Contrôle	Commande / action	GO si...
1	Inventaire des fichiers data	Lister config, rules, templates, static	Ressources identifiées
2	Inclusion explicite	`--include-data-files` / `--include-data-dir`	Build contient les ressources
3	Imports dynamiques	`--include-module` / `--include-package`	Aucun `ModuleNotFoundError`

#	Contrôle	Commande / action	GO si...
4	Chemins robustes	Éviter chemins relatifs fragiles	Fonctionne hors repo
5	Standalone inspecté	Ouvrir `.dist`	Fichiers présents
6	DLL / SO vérifiées	`ldd`, listing DLL, test machine cible	Aucune lib manquante
7	Django templates	`get_template(...)`	Template trouvé
8	Django static	`findstatic`	Static trouvé
9	Test hors dossier source	Copie dans `/tmp` ou `C:\temp`	Application démarre
10	Inspection release	Vérifier absence `.env`, tests, sources sensibles	Livrable propre

---

Commande complète de référence

```
python -m nuitka \
                        --mode=standalone \
                        --output-dir=build/nuitka \
                        --remove-output \
                        --show-progress \
                        --report=build/reports/resources-report.xml \
                        --assume-yes-for-downloads \
                        --include-data-files=config/defaults.yaml=config/defaults.yaml \
                        --include-data-files=rules/*.json=rules/ \
                        --include-data-dir=templates=templates \
                        --include-data-dir=static=static \
                        --include-module=my_package.dynamic_plugin \
                        --include-package=my_package.rules \
                        app.py
```

> **Conclusion :** un build Nuitka professionnel n'est pas seulement un exécutable. C'est un ensemble cohérent : code compilé, ressources présentes, imports dynamiques déclarés, dépendances natives disponibles, chemins runtime robustes et livrable nettoyé.

5.2 Protection du code — protection raisonnable, limites, architecture hybride

Protection du code : être ambitieux, mais lucide

Nuitka améliore fortement la protection pratique d'un code Python distribué, parce qu'il évite de livrer directement les fichiers `.py` du cœur métier. Mais il ne faut pas présenter cela comme une protection absolue ou comme une obfuscation inviolable. Le bon vocabulaire est : **protection raisonnable**, **barrière technique**, **réduction de lisibilité**, **augmentation du coût de reverse engineering**.

Ce que l'on cherche vraiment à protéger

```
Valeur métier à protéger
                                │
                                ├── Algorithmes propriétaires
                                │   ├── scoring
                                │   ├── heuristiques
                                │   ├── règles d'analyse
                                │   └── classification
                                │
                                ├── Savoir-faire technique
                                │   ├── moteur de diagnostic
                                │   ├── introspection avancée
                                │   ├── détection de patterns
                                │   └── génération de recommandations
                                │
                                ├── Avantage concurrentiel
                                │   ├── méthodes internes
                                │   ├── pondérations
                                │   ├── workflows avancés
                                │   └── logique de priorisation
                                │
                                └── Ce qu'il ne faut jamais protéger uniquement par compilation
                                ├── secrets
                                ├── clés API
                                ├── tokens clients
                                └── licence maître
```

Positionnement professionnel : Nuitka ne rend pas le reverse engineering impossible, mais rend le vol direct du code source beaucoup moins immédiat.

Résumé clair

Question	Réponse réaliste
Nuitka cache-t-il le code source ?	Oui, les modules compilés ne sont plus distribués comme `.py`.
Est-ce de l'obfuscation parfaite ?	Non. Un binaire reste analysable par des moyens avancés.
Est-ce mieux que livrer du `.py` ?	Oui, très nettement.
Est-ce mieux que du `.pyc` ?	Oui, car le bytecode Python est généralement plus facile à décompiler.
Peut-on y mettre des secrets ?	Non. Jamais de secrets codés en dur.
Meilleur usage ?	Compiler le cœur métier, pas toute l'application.

Formulation recommandée : “Nuitka protège raisonnablement notre propriété intellectuelle en transformant le cœur métier Python en extension binaire, tout en conservant une API Python stable.”

Niveaux de protection : du .py au module compilé

Toutes les formes de distribution Python n'offrent pas le même niveau de protection. Le niveau réel dépend du type de livrable, de ce que vous incluez dans le package, de la présence ou non des sources, et de l'architecture générale.

Niveau	Distribution	Lisibilité	Protection pratique	Usage typique
0	Fichiers `.py`	Totale	Très faible	Open source, interne, dev.
1	`.pyc`	Décompilable	Faible	Distribution naïve.
2	Obfuscation Python	Réduite	Moyenne	Barrière légère.
3	Nuitka module `.pyd` / `.so`	Faible	Bonne	Protection d'un moteur métier.
4	Nuitka + architecture hybride + release nettoyée	Très faible	Très bonne en pratique	Addon commercial / produit fermé.
5	Architecture serveur / SaaS	Non livré au client	Meilleure protection	API distante, produit cloud.

Évolution recommandée

```
Étape 1
                                Source Python interne
                                |
                                ▼
                                Étape 2
                                Identifier le cœur métier
                                |
                                ▼
                                Étape 3
                                Créer une API publique stable
                                |
                                ▼
                                Étape 4
                                Compiler le cœur avec Nuitka
                                |
                                ▼
                                Étape 5
                                Retirer les sources sensibles du livrable
                                |
                                ▼
                                Étape 6
                                Tester la wheel / release propre
                                |
                                ▼
                                Étape 7
                                Ajouter licence, contrat, activation si nécessaire
```

Erreur fréquente

Compiler ne suffit pas : si le package final contient encore `scoring.py` à côté de `scoring.pyd`, alors la protection du module compilé est largement annulée.

Livrable propre

```
release_package/
                                    │
                                    ├── public Python layer
                                    │   ├── admin.py
                                    │   ├── views.py
                                    │   └── core_api.py
                                    │
                                    ├── protected binary layer
                                    │   ├── scoring.pyd / scoring.so
                                    │   ├── analyzer.pyd / analyzer.so
                                    │   └── rules_engine.pyd / rules_engine.so
                                    │
                                    └── resources
                                    ├── templates/
                                    ├── static/
                                    └── safe rules/config
```

Menaces réalistes : contre quoi se protège-t-on ?

La protection du code doit partir d'un modèle de menace réaliste. On ne protège pas de la même façon contre un utilisateur curieux, un concurrent pressé, un développeur interne, un client technique, ou un attaquant spécialisé en reverse engineering.

Profil	Capacité	Risque	Effet de Nuitka	Mesure complémentaire
Utilisateur curieux	Ouvre les fichiers livrés	Lecture directe des `.py`	Très efficace	Ne pas livrer les sources sensibles.
Développeur Python	Inspecte package, imports, bytecode	Compréhension rapide du moteur	Efficace	API publique limitée, package nettoyé.
Concurrent pressé	Copie les sources si disponibles	Clonage fonctionnel	Très utile	Licence, mentions légales, preuves d'antériorité.
Client technique avancé	Analyse binaire basique	Compréhension partielle	Barrière moyenne à forte	Contrat, activation, architecture serveur si besoin.
Reverse engineer motivé	Outils binaires avancés	Analyse profonde	Barrière, pas blocage absolu	Ne pas livrer les secrets, déplacer la logique critique serveur.

Menaces réduites par Nuitka

- Copie directe des fichiers `.py` sensibles.
- Lecture rapide des heuristiques internes.
- Décompilation simple du bytecode Python.
- Modification naïve de fonctions métier.
- Distribution d'une copie quasi identique du moteur.

Menaces non résolues par Nuitka seul

- Analyse binaire avancée par personne motivée.
- Extraction de chaînes texte présentes dans le binaire.
- Vol de secrets codés en dur.
- Contournement logique si la licence est uniquement locale.
- Copie de l'interface publique et imitation du comportement.

Conclusion menace : Nuitka est excellent pour réduire la copie facile. Pour protéger des secrets ou une logique extrêmement critique, il faut compléter avec serveur, licence, contrat, ou architecture d'activation.

Architecture hybride : public layer + protected core

La meilleure stratégie consiste à séparer ce qui doit rester maintenable, intégrable et visible de ce qui constitue réellement le cœur propriétaire. Cette architecture permet de garder un projet propre tout en protégeant les modules sensibles.

Architecture recommandée

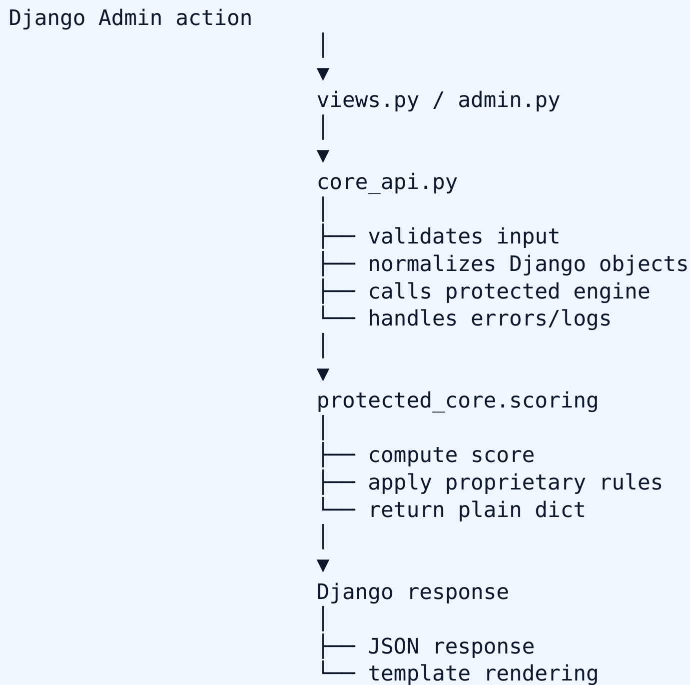
```
Public layer
    │
    ├── admin.py
    ├── views.py
    ├── urls.py
    ├── apps.py
    ├── templates/
    ├── static/
    └── core_api.py
    │
    │ calls stable API
    ▼
Protected layer
    │
    ├── scoring.pyd / scoring.so
    ├── analyzer.pyd / analyzer.so
    ├── rules_engine.pyd / rules_engine.so
    └── metadata_builder.pyd / metadata_builder.so
    │
    │ reads safe resources
    ▼
Resources
    │
    ├── public templates
    ├── public static assets
    └── non-secret rules/config
```

Responsabilités par couche

Couche	Responsabilité	Protection
Public layer	Intégration Django, UI, endpoints, routing	Lisible et maintenable
API wrapper	Contrat stable entre Django et core	Source lisible, logique minimale
Protected core	Algorithmes, scoring, heuristiques	Compilé Nuitka
Resources	Templates, static, règles non secrètes	Livrées explicitement
External control	Licence, activation, contrat	Hors binaire si possible

Bon design : la couche publique ne contient que de la glue logic. La valeur métier est concentrée dans des modules courts, stables et compilables.

Flux d'appel propre



```
Django Admin action
                        │
                        ▼
                        views.py / admin.py
                        │
                        ▼
                        core_api.py
                        │
                        ├── validates input
                        ├── normalizes Django objects
                        ├── calls protected engine
                        └── handles errors/logs
                        │
                        ▼
                        protected_core.scoring
                        │
                        ├── compute score
                        ├── apply proprietary rules
                        └── return plain dict
                        │
                        ▼
                        Django response
                        │
                        ├── JSON response
                        └── template rendering
```

Ce que Nuitka apporte concrètement

Nuitka apporte une protection technique pragmatique. Il transforme les modules ciblés en extensions binaires et évite la distribution directe du source Python sensible. Cela améliore également la perception professionnelle du livrable.

Bénéfices directs

- Suppression de la distribution directe des fichiers `.py` pour les modules compilés.
- Barrière technique plus forte que du bytecode Python `.pyc`.
- Packaging plus professionnel pour un addon distribué.
- Possibilité de garder une interface Python lisible et de cacher le noyau.
- Réduction du risque de copie immédiate des algorithmes.
- Compatible avec une architecture progressive : un module à la fois.

Commande typique

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                protected_core/scoring.py
```

Avant / après

Avant	Après Nuitka
`scoring.py` lisible	`scoring.pyd` / `scoring.so`
Algorithme visible	Algorithme non lisible directement
Copie facile	Copie plus difficile à exploiter
Modification simple	Modification beaucoup moins directe

Avant	Après Nuitka
Package source complet	Package hybride public + binaire

Usage idéal : compiler des fichiers qui contiennent une vraie valeur métier : scoring, analyse, règles, diagnostic, parsing, déduplication, heuristiques.

Limites : ce que Nuitka ne garantit pas

Une bonne stratégie de protection doit expliciter ses limites. Cela évite les fausses promesses et permet d'ajouter les bonnes protections complémentaires lorsque nécessaire.

Limite	Réalité	Réponse recommandée
Reverse engineering	Un binaire peut toujours être analysé.	Accepter que Nuitka augmente le coût, pas qu'il rend impossible.
Chaînes texte	Certains textes peuvent apparaître dans le binaire.	Ne pas stocker secrets, messages internes sensibles, clés.
API publique	Les fonctions appelées restent compréhensibles par usage.	Limiter l'API publique à un contrat minimal.
Secrets	Un secret codé en dur peut être extrait.	Utiliser variables d'environnement, serveur ou coffre.
Licence locale	Peut être contournée si tout est local.	Ajouter activation serveur si enjeu élevé.
Compatibilité	Les extensions dépendent OS/Python/architecture.	Builder par plateforme cible.
Sources livrées	Si les `.py` sensibles restent dans la wheel, protection réduite.	Nettoyer le package final.

Ce qui reste exposé

- Les noms publics appelés depuis la couche Python.
- Certains messages d'erreur ou chaînes de caractères.
- Les comportements observables du moteur.
- Les endpoints et interfaces visibles.
- Les fichiers data livrés comme JSON/YAML.

Comment réduire l'exposition

- Limiter l'API publique.
- Éviter les noms trop révélateurs dans les fonctions publiques.
- Ne pas mettre de secrets dans le code.
- Nettoyer la release avant livraison.
- Déplacer les décisions très critiques côté serveur.

Message à retenir : Nuitka protège très bien contre la lecture et copie faciles. Il ne remplace ni la sécurité des secrets, ni la licence commerciale, ni une architecture serveur pour les fonctions ultra-critiques.

Secrets, licences, activation : ne pas confondre protection du code et gestion des droits

Une erreur fréquente consiste à croire qu'un secret placé dans un module compilé devient invisible. C'est dangereux. Un binaire peut être inspecté. Les secrets doivent être gérés hors du code, via variables d'environnement, fichiers de configuration sécurisés, serveur d'activation, coffre de secrets ou mécanisme contractuel.

À ne jamais faire

```
# Mauvais exemple : secret codé en dur
                                MASTER_LICENSE_KEY = "ABC-SECRET-PRIVATE-KEY"

                                API_TOKEN = "sk_live_xxxxxxxxxxxxxxxx"

                                def validate_license(user_key: str) -> bool:
                                return user_key == MASTER_LICENSE_KEY
```

Danger : compiler ce code ne rend pas ces secrets sûrs. Il ne faut jamais coder en dur une clé maître ou un token client.

Approches plus sûres

Besoin	Approche recommandée
Clé API	Variable d'environnement ou coffre de secrets.
Licence client	Fichier licence signé, pas secret maître local.
Activation	Serveur d'activation si enjeu commercial fort.
Contrôle d'accès	Compte client, entitlement, contrat.
Logique ultra-sensible	Service distant / API SaaS.

Pattern licence locale raisonnable

```
Client installation
                                │
                                ├── compiled addon
                                ├── public license file
                                │   └── signed entitlement
                                │
                                └── local verifier
                                ├── checks signature
                                ├── checks expiry
                                └── never contains master private key
```

Règles de sécurité simples

Règle	Pourquoi
Ne jamais coder en dur une clé secrète.	Un binaire peut être analysé.
Ne jamais livrer `.env`.	Contient souvent des mots de passe.
Ne jamais stocker une clé privée de signature dans l'addon.	Elle permettrait de fabriquer de fausses licences.
Utiliser une clé publique côté client si signature nécessaire.	La clé publique peut être distribuée.
Mettre les décisions critiques côté serveur si possible.	Ce qui n'est pas livré est beaucoup mieux protégé.

Cas Django addon : protéger ideolab_admin_tools

Pour `ideolab_admin_tools`, l'approche la plus solide consiste à compiler le cœur de scoring / analyse / règles, tout en gardant l'intégration Django claire. Cela permet de publier ou distribuer un addon maintenable sans exposer directement les algorithmes différenciants.

Structure recommandée

```
ideolab_admin_tools/
                                │
                                ├── public layer
                                │   ├── apps.py
                                │   ├── admin.py
                                │   ├── views.py
                                │   ├── urls.py
                                │   ├── templates/
                                │   └── static/
                                │
                                ├── bridge
                                │   ├── core_api.py
                                │   ├── dto.py
                                │   └── normalizers.py
                                │
                                └── protected_core/
                                ├── scoring.pyd / scoring.so
                                ├── analyzer.pyd / analyzer.so
                                ├── rules_engine.pyd / rules_engine.so
                                └── metadata_builder.pyd / metadata_builder.so
```

Ce qui doit rester public

Fichier	Pourquoi le garder lisible
`apps.py`	Déclaration Django standard.
`admin.py`	Intégration avec Django Admin.
`views.py`	HTTP, permissions, endpoints.
`templates/`	HTML visible par nature.
`core_api.py`	Contrat stable avec le moteur compilé.

Ce qui doit être protégé

- Scoring avancé des ModelAdmin.
- Heuristiques de diagnostic.
- Règles de classification des problèmes.
- Analyse d'introspection propriétaire.
- Génération de recommandations.

Commande de compilation type

```
python -m nuitka \
                        --mode=module \
                        --output-dir=build/nuitka_modules \
                        --remove-output \
                        --show-progress \
                        --report=build/reports/scoring-report.xml \
                        ideolab_admin_tools/protected_core/scoring.py
```

> **Premier objectif réaliste :** compiler `protected_core/scoring.py`, tester l'import dans Django, puis élargir vers `analyzer.py` et `rules_engine.py`.

Release protégée : nettoyer ce qui est livré

La protection réelle dépend énormément du contenu final de la release. Même avec Nuitka, une mauvaise wheel ou archive peut livrer par accident des fichiers sensibles, tests internes, sources propriétaires, secrets ou documentation privée.

Pipeline de release sécurisé

```
Release sécurisée
                                │
                                ├── 1. Tests source internes
                                ├── 2. Build Nuitka modules
                                ├── 3. Copie .pyd / .so
                                ├── 4. Suppression sources sensibles du livrable
                                ├── 5. Inclusion templates/static nécessaires
                                ├── 6. Exclusion .env, tests, docs internes
                                ├── 7. Build wheel / archive
                                ├── 8. Installation dans venv propre
                                ├── 9. Test import compiled core
                                ├── 10. Test Django admin
                                └── 11. Inspection manuelle du contenu final
```

À exclure impérativement

Élément	Risque
`.env`	Secrets, tokens, mots de passe.
`.git/`	Historique complet.
`tests/`	Révèle les comportements attendus.
`docs/internal/`	Architecture privée.
`protected_core/*.py` sensibles	Annule l'intérêt du module compilé.
`dev_config.*`	Chemins locaux et réglages internes.

> **Contrôle obligatoire :** toujours ouvrir la wheel ou l'archive finale pour vérifier son contenu avant livraison.

Commandes d'inspection

```
# Voir le contenu d'une wheel
                        python -m zipfile -l dist/ideolab_admin_tools-*.whl

                        # Chercher des fichiers sensibles dans une release décompressée
                        find release/ -name ".env" -o -name "*.py" -o -path "*/tests/*" -o -path "*/docs/
internal/*"

                        # Vérifier import dans un venv propre
                        python -m venv /tmp/ideolab_release_test
                        source /tmp/ideolab_release_test/bin/activate
                        python -m pip install dist/ideolab_admin_tools-*.whl
                        python -c "import ideolab_admin_tools.protected_core.scoring; print('compiled core
OK')"
```

Checklist finale : protection raisonnable avec Nuitka

#	Contrôle	GO si...	NO-GO si...
1	Cœur métier identifié	Modules sensibles listés clairement	Tout le projet est considéré sensible
2	API publique stable	Django appelle `core_api.py`	Django appelle directement les détails internes
3	Modules compilés	`.pyd` / `.so` générés	Seulement des `.py` livrés
4	Sources sensibles retirées	Pas de `protected_core/*.py` sensible dans release	Sources sensibles présentes
5	Secrets absents	Pas de `.env`, token, clé maître	Secret codé en dur ou livré
6	Fallback maîtrisé	Autorisé dev, interdit prod	Fallback Python actif en production
7	Tests runtime	Import compilé OK dans venv propre	Fonctionne seulement dans le repo
8	Package inspecté	Wheel / archive vérifiée	Contenu final inconnu
9	Licence / contrat	Conditions d'usage claires	Protection uniquement technique
10	Message commercial réaliste	“Protection raisonnable”	“Inviolable” ou “impossible à reverse-engineer”

Phrase recommandée

Formulation : “Le cœur métier est distribué sous forme de modules compilés, ce qui réduit fortement l'exposition du code source et rend la copie directe beaucoup plus difficile, tout en gardant une intégration Django maintenable.”

Phrase à éviter

À éviter : “Le code est impossible à récupérer.” C'est trop absolu et techniquement imprudent.

Conclusion

```
Bonne protection Nuitka
                        │
                        ├── petit cœur métier isolé
                        ├── API publique minimale
                        ├── modules compilés
                        ├── sources sensibles absentes
                        ├── secrets hors code
                        ├── tests en venv propre
                        ├── packaging inspecté
                        └── licence / contrat / activation si besoin
```

6.1 Build industriel Nuitka — scripts, Makefile, PowerShell, CI/CD, releases versionnées

Pourquoi industrialiser un build Nuitka ?

Un build Nuitka manuel est acceptable pour un premier test. Mais dès que l'on veut distribuer un outil, un addon Django, un module propriétaire ou un exécutable client, il faut rendre le build **reproductible**, **traçable**, **testé**, **versionné** et **archivable**. Sinon, on ne sait plus quelle commande a produit quel binaire, avec quelle version de Python, quelle version de Nuitka, quelles options, quels fichiers inclus, et quels tests ont été exécutés.

Pipeline industriel cible

```
Developer / CI runner
                                │
                                ├── 1. Clean workspace
                                ├── 2. Create / activate venv
                                ├── 3. Install build requirements
                                ├── 4. Run source tests
                                ├── 5. Run Django checks if needed
                                ├── 6. Build with Nuitka
                                ├── 7. Capture logs and reports
                                ├── 8. Run binary smoke tests
                                ├── 9. Package artefacts
                                ├── 10. Compute hashes
                                ├── 11. Upload artefacts
                                └── 12. Tag release
```

Règle professionnelle : un binaire ne doit jamais être produit par une commande improvisée dans un terminal. Il doit être produit par un script versionné.

Ce qu'un build industriel doit produire

Élément	Rôle	Exemple
Artefact	Livrable final	`.exe`, `.dist.zip`, `.whl`
Build log	Diagnostic	`build/logs/build.log`
Report Nuitka	Analyse compilation	`compilation-report.xml`
Environment report	Reproductibilité	Python, OS, Nuitka, compiler
Validation report	GO / NO-GO	Tests source + binaire
Checksums	Intégrité	`SHA256SUMS.txt`

Attention : un build qui marche sur une machine développeur mais qui n'est pas reproductible en CI n'est pas encore industrialisé.

Arborescence recommandée pour un projet avec builds Nuitka

Une structure claire évite de mélanger sources, artefacts, logs, rapports, scripts et releases. Elle facilite aussi l'automatisation CI/CD et l'analyse des échecs.

Structure projet

```
project/
                                │
                                ├── app.py
                                ├── pyproject.toml
                                ├── requirements.txt
                                ├── requirements-build.txt
                                │
                                ├── src/
                                │   └── my_package/
                                │
                                ├── tests/
                                │   ├── test_source.py
                                │   ├── test_binary.py
                                │   └── test_packaging.py
                                │
                                ├── scripts/
                                │   ├── build_nuitka.py
                                │   ├── build_windows.ps1
                                │   ├── build_linux.sh
                                │   ├── clean_build.py
                                │   └── validate_artifact.py
                                │
                                ├── build/
                                │   ├── nuitka/
                                │   ├── logs/
                                │   ├── reports/
                                │   └── tmp/
                                │
                                ├── dist/
                                │   ├── releases/
                                │   └── wheels/
                                │
                                └── .github/
                                └── workflows/
                                └── nuitka-build.yml
```

Rôle des dossiers

Dossier	Rôle	Versionné ?
`scripts/`	Scripts de build et validation	Oui
`tests/`	Tests source, binaire, packaging	Oui
`build/`	Sorties temporaires et logs locaux	Non
`dist/`	Artefacts finaux	Non ou release seulement
`.github/workflows/`	CI/CD	Oui

Exemple `.gitignore`

```
# Build outputs
                                build/
                                dist/
                                *.build/
                                *.dist/
                                *.onefile-build/

                                # Python
                                __pycache__/
                                *.pyc
                                .venv/

                                # Secrets
                                .env
                                *.key
                                *.pem

                                # Nuitka local reports
                                *.log
                                *.xml
```

Bonne pratique : versionner les scripts et workflows, mais pas les artefacts générés localement. Les artefacts finaux doivent être attachés à une release ou stockés dans un dépôt d'artefacts.

Makefile Linux / macOS

Le Makefile permet d'encapsuler les commandes longues en cibles simples : `make test`, `make build-standalone`, `make build-onefile`, `make release`. C'est très pratique pour Linux, macOS et serveurs CI.

Makefile complet

```
PYTHON ?= python
                                APP ?= app.py
                                APP_NAME ?= app
                                OUT ?= build/nuitka
                                LOGS ?= build/logs
                                REPORTS ?= build/reports
                                DIST ?= dist/releases
                                JOBS ?= 8

                                .PHONY: help clean env test check build-standalone build-onefile validate
package release


                                help:
                                @echo "Available targets:"
                                @echo "  make clean"
                                @echo "  make env"
                                @echo "  make test"
                                @echo "  make build-standalone"
                                @echo "  make build-onefile"
                                @echo "  make validate"
                                @echo "  make release"

                                clean:
                                rm -rf $(OUT) $(LOGS) $(REPORTS)
                                rm -rf *.build *.dist *.onefile-build
                                mkdir -p $(OUT) $(LOGS) $(REPORTS) $(DIST)

                                env:
                                $(PYTHON) -m pip install --upgrade pip setuptools wheel
                                $(PYTHON) -m pip install -r requirements-build.txt

                                test:
                                $(PYTHON) -m pytest -v

                                check:
                                $(PYTHON) -m compileall .

                                build-standalone: clean test
                                $(PYTHON) -m nuitka \
                                --mode=standalone \
                                --output-dir=$(OUT) \
                                --remove-output \
                                --jobs=$(JOBS) \
                                --show-progress \
                                --report=$(REPORTS)/standalone-report.xml \
                                --assume-yes-for-downloads \
                                $(APP) 2>&1 | tee $(LOGS)/standalone-build.log

                                build-onefile: clean test
                                $(PYTHON) -m nuitka \
                                --mode=onefile \
                                --output-dir=$(OUT) \
                                --remove-output \
                                --jobs=$(JOBS) \
                                --show-progress \
                                --report=$(REPORTS)/onefile-report.xml \
                                --assume-yes-for-downloads \
                                $(APP) 2>&1 | tee $(LOGS)/onefile-build.log

                                validate:
                                $(PYTHON) scripts/validate_artifact.py

                                package:
                                cd $(OUT) && zip -r ../../$(DIST)/$(APP_NAME)-standalone.zip $
(APP_NAME).dist


                                release: build-standalone validate package
                                sha256sum $(DIST)/* > $(DIST)/SHA256SUMS.txt
                                @echo "Release ready in $(DIST)"
```

Commandes d'utilisation

```
# Installer l'environnement de build
                                make env

                                # Lancer les tests
                                make test

                                # Compiler standalone
                                make build-standalone

                                # Compiler onefile
                                make build-onefile

                                # Valider l'artefact
                                make validate

                                # Construire une release complète
                                make release
```

Variables utiles

Variable	Usage
`PYTHON`	Choisir l'exécutable Python.
`APP`	Point d'entrée à compiler.
`APP_NAME`	Nom du livrable.
`OUT`	Dossier de sortie Nuitka.
`JOBS`	Nombre de jobs compilation.

Attention : Makefile exige des tabulations réelles avant les commandes, pas des espaces.

PowerShell Windows : build reproductible

Sous Windows, un script PowerShell est souvent plus lisible qu'une longue commande copiée-collée. Il permet aussi de stopper le build à la première erreur et de centraliser les logs.

Script `scripts/build_windows.ps1`

```
param(
                                [string]$App = "app.py",
                                [string]$AppName = "app",
                                [string]$OutputDir = "build\nuitka",
                                [string]$LogsDir = "build\logs",
                                [string]$ReportsDir = "build\reports",
                                [int]$Jobs = 8,
                                [switch]$Onefile
                                )

                                $ErrorActionPreference = "Stop"

                                Write-Host "=== Nuitka Windows Build ===" -ForegroundColor Cyan
                                Write-Host "App: $App"
                                Write-Host "Mode: $(if ($Onefile) { 'onefile' } else { 'standalone' })"

                                New-Item -ItemType Directory -Force $OutputDir, $LogsDir, $ReportsDir |
Out-Null


                                Write-Host "Installing build requirements..." -ForegroundColor Yellow
                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -r requirements-build.txt

                                Write-Host "Running tests..." -ForegroundColor Yellow
                                python -m pytest -v

                                $Mode = if ($Onefile) { "onefile" } else { "standalone" }
                                $Report = Join-Path $ReportsDir "$Mode-report.xml"
                                $Log = Join-Path $LogsDir "$Mode-build.log"

                                Write-Host "Building with Nuitka..." -ForegroundColor Yellow

                                $Arguments = @(
                                "-m", "nuitka",
                                "--mode=$Mode",
                                "--output-dir=$OutputDir",
                                "--remove-output",
                                "--jobs=$Jobs",
                                "--show-progress",
                                "--report=$Report",
                                "--assume-yes-for-downloads",
                                $App
                                )

                                python @Arguments *>&1 | Tee-Object $Log

                                if ($LASTEXITCODE -ne 0) {
                                throw "Nuitka build failed with exit code $LASTEXITCODE"
                                }

                                Write-Host "Build completed successfully." -ForegroundColor Green
                                Write-Host "Log: $Log"
                                Write-Host "Report: $Report"
```

Utilisation

```
# Standalone
                                .\scripts\build_windows.ps1 `
                                -App "app.py" `
                                -AppName "app" `
                                -Jobs 8

                                # Onefile
                                .\scripts\build_windows.ps1 `
                                -App "app.py" `
                                -AppName "app" `
                                -Jobs 8 `
                                -Onefile
```

Script de validation Windows

```
# scripts/validate_windows.ps1
                                $ErrorActionPreference = "Stop"

                                $Binary = "build\nuitka\app.dist\app.exe"

                                if (!(Test-Path $Binary)) {
                                throw "Binary not found: $Binary"
                                }

                                & $Binary

                                if ($LASTEXITCODE -ne 0) {
                                throw "Binary failed with exit code $LASTEXITCODE"
                                }

                                Write-Host "Binary validation OK" -ForegroundColor Green
```

Conseil Windows : utiliser PowerShell pour orchestrer, mais garder les options Nuitka identiques à celles utilisées en CI.

Script Python de build multi-plateforme

Un script Python est souvent le meilleur compromis pour un projet Python : il fonctionne sous Windows, Linux et macOS, peut produire des logs, vérifier l'environnement, lancer pytest, appeler Nuitka et générer un rapport de validation.

Script `scripts/build_nuitka.py`

```
from __future__ import annotations

                                import argparse
                                import platform
                                import shutil
                                import subprocess
                                import sys
                                from pathlib import Path


                                ROOT = Path(__file__).resolve().parent.parent
                                BUILD_DIR = ROOT / "build"
                                NUITKA_DIR = BUILD_DIR / "nuitka"
                                LOGS_DIR = BUILD_DIR / "logs"
                                REPORTS_DIR = BUILD_DIR / "reports"


                                def run(command: list[str], log_file: Path | None = None) -> None:
                                print("+", " ".join(command))

                                if log_file is None:
                                result = subprocess.run(command, cwd=ROOT)
                                if result.returncode != 0:
                                raise SystemExit(result.returncode)
                                return

                                with log_file.open("w", encoding="utf-8") as stream:
                                process = subprocess.Popen(
                                command,
                                cwd=ROOT,
                                stdout=subprocess.PIPE,
                                stderr=subprocess.STDOUT,
                                text=True,
                                )

                                assert process.stdout is not None

                                for line in process.stdout:
                                print(line, end="")
                                stream.write(line)

                                code = process.wait()
                                if code != 0:
                                raise SystemExit(code)


                                def clean() -> None:
                                for path in [NUITKA_DIR, LOGS_DIR, REPORTS_DIR]:
                                if path.exists():
                                shutil.rmtree(path)
                                path.mkdir(parents=True, exist_ok=True)


                                def write_environment_report() -> None:
                                REPORTS_DIR.mkdir(parents=True, exist_ok=True)

                                content = [
                                f"platform={platform.platform()}",
                                f"machine={platform.machine()}",
                                f"python={sys.version}",
                                f"executable={sys.executable}",
                                ]

                                (REPORTS_DIR / "environment.txt").write_text("\n".join(content),
encoding="utf-8")


                                def main() -> int:
                                parser = argparse.ArgumentParser()
                                parser.add_argument("--app", default="app.py")
                                parser.add_argument("--mode", choices=["standalone", "onefile", "module"],
default="standalone")

                                parser.add_argument("--jobs", default="8")
```

```
parser.add_argument("--skip-tests", action="store_true")
args = parser.parse_args()

clean()
write_environment_report()

run([sys.executable, "-m", "pip", "install", "-r", "requirements-
build.txt"])


if not args.skip_tests:
run([sys.executable, "-m", "pytest", "-v"])

report = REPORTS_DIR / f"{args.mode}-report.xml"
log = LOGS_DIR / f"{args.mode}-build.log"

command = [
sys.executable,
"-m",
"nuitka",
f"--mode={args.mode}",
f"--output-dir={NUITKA_DIR}",
"--remove-output",
f"--jobs={args.jobs}",
"--show-progress",
f"--report={report}",
"--assume-yes-for-downloads",
args.app,
]

run(command, log_file=log)

print("Build OK")
print(f"Log: {log}")
print(f"Report: {report}")

return 0


if __name__ == "__main__":
raise SystemExit(main())
```

Utilisation

```
# Build standalone
python scripts/build_nuitka.py --app app.py --mode standalone

# Build onefile
python scripts/build_nuitka.py --app app.py --mode onefile

# Build module
python scripts/build_nuitka.py \
--app ideolab_admin_tools/protected_core/scoring.py \
--mode module

# Build sans relancer pytest
python scripts/build_nuitka.py --app app.py --mode standalone --skip-tests
```

Avantages du script Python

Avantage	Impact
Multi-plateforme	Windows, Linux, macOS.
Versionnable	Historique clair des options.
Extensible	Ajout facile de checks, logs, packaging.
Lisible	Plus clair qu'une commande de 20 lignes.
Compatible CI	Un seul entrypoint de build.

> **Pour IDEO-Lab :** un script Python de build est souvent le meilleur choix, car il reste dans l'écosystème Python/Django et peut intégrer des checks métier.

GitHub Actions minimal : build Nuitka en CI

GitHub Actions permet de valider que le build fonctionne hors de la machine développeur. Le workflow ci-dessous installe Python, installe les dépendances de build, lance les tests, compile avec Nuitka, puis upload les artefacts.

Workflow standalone Windows

```
name: nuitka-build-windows

on:
push:
branches: ["main"]
pull_request:
workflow_dispatch:

jobs:
build-windows:
runs-on: windows-latest

steps:
- name: Checkout
uses: actions/checkout@v4

- name: Setup Python
uses: actions/setup-python@v5
with:
python-version: "3.12"

- name: Install build dependencies
run: |
python -m pip install --upgrade pip setuptools wheel
python -m pip install -r requirements-build.txt

- name: Run tests
run: |
python -m pytest -v

- name: Build with Nuitka
run: |
python -m nuitka `
--mode=standalone `
--output-dir=build\nuitka `
--remove-output `
--show-progress `
--report=build\reports\compilation-report.xml `
--assume-yes-for-downloads `
app.py

- name: Upload Nuitka artifact
uses: actions/upload-artifact@v4
with:
name: app-windows-standalone
path: |
build/nuitka/**/*.dist/**
build/reports/**
build/logs/**
if-no-files-found: error
```

Workflow Linux standalone

```
name: nuitka-build-linux

                                on:
                                push:
                                branches: ["main"]
                                pull_request:
                                workflow_dispatch:

                                jobs:
                                build-linux:
                                runs-on: ubuntu-latest

                                steps:
                                - name: Checkout
                                uses: actions/checkout@v4

                                - name: Install system packages
                                run: |
                                sudo apt update
                                sudo apt install -y build-essential ccache patchelf zip

                                - name: Setup Python
                                uses: actions/setup-python@v5
                                with:
                                python-version: "3.12"

                                - name: Install build dependencies
                                run: |
                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -r requirements-build.txt

                                - name: Run tests
                                run: |
                                python -m pytest -v

                                - name: Build with Nuitka
                                run: |
                                mkdir -p build/logs build/reports
                                python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/compilation-report.xml \
                                --assume-yes-for-downloads \
                                app.py 2>&1 | tee build/logs/build.log

                                - name: Upload Nuitka artifact
                                uses: actions/upload-artifact@v4
                                with:
                                name: app-linux-standalone
                                path: |
                                build/nuitka/**/*.dist/**
                                build/reports/**
                                build/logs/**
                                if-no-files-found: error
```

Important : les artefacts Nuitka sont dépendants de l'OS, de l'architecture et de la version Python. Un binaire Windows doit être buildé sur Windows, un binaire Linux sur Linux.

Matrix CI : plusieurs OS et versions Python

Si le produit doit être distribué à plusieurs environnements, il faut produire des artefacts séparés. Une matrix GitHub Actions permet de builder Windows/Linux/macOS et plusieurs versions Python.

Workflow matrix

```
name: nuitka-matrix-build

                                on:
                                workflow_dispatch:
                                push:
                                tags:
                                - "v*"

                                jobs:
                                build:
                                strategy:
                                fail-fast: false
                                matrix:
                                os: [windows-latest, ubuntu-latest]
                                python-version: ["3.11", "3.12"]

                                runs-on: $

                                steps:
                                - uses: actions/checkout@v4

                                - name: Install Linux system packages
                                if: runner.os == 'Linux'
                                run: |
                                sudo apt update
                                sudo apt install -y build-essential ccache patchelf zip

                                - uses: actions/setup-python@v5
                                with:
                                python-version:  matrix.python-version

                                - name: Install dependencies
                                run: |
                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -r requirements-build.txt

                                - name: Run tests
                                run: |
                                python -m pytest -v

                                - name: Build with Python script
                                run: |
                                python scripts/build_nuitka.py --app app.py --mode standalone

                                - name: Upload artifact
                                uses: actions/upload-artifact@v4
                                with:
                                name: app-$-py$ matrix.python-version
                                path: |
                                build/nuitka/**
                                build/reports/**
                                build/logs/**
                                if-no-files-found: error
```

Résultat attendu

```
CI artifacts
│
├── app-Windows-py3.11
├── app-Windows-py3.12
├── app-Linux-py3.11
└── app-Linux-py3.12
```

Pourquoi plusieurs artefacts ?

Dimension	Impact
OS	Windows `.exe/.pyd`, Linux `.bin/.so`.
Python	ABI différente pour modules compilés.
Architecture	x86_64, ARM64, etc.
Dépendances natives	DLL / SO différentes.

Pour un addon Django compilé : publier clairement les roues ou extensions par OS et version Python cible.

Gestion des artefacts : nommage, version, hash, logs

Les artefacts doivent être nommés de façon à identifier clairement leur contenu. Un fichier `app.zip` ne suffit pas. Il faut indiquer version, OS, architecture, Python, mode Nuitka et éventuellement commit.

Convention de nommage

```
dist/releases/
│
├── ideolab-admin-tools-0.1.0-windows-amd64-py312-module.zip
├── ideolab-admin-tools-0.1.0-linux-x86_64-py312-module.zip
├── ideolab-cli-0.1.0-windows-amd64-py312-standalone.zip
├── ideolab-cli-0.1.0-linux-x86_64-py312-standalone.zip
├── SHA256SUMS.txt
├── RELEASE_NOTES.md
└── validation-report.md
```

Métadonnées à conserver

Métadonnée	Exemple
Version produit	`0.1.0`
Commit Git	`abc1234`
OS	`windows-latest`, `ubuntu-latest`
Python	`3.12.4`
Nuitka	Version exacte
Build mode	`standalone`, `onefile`, `module`
Checksum	SHA256

Créer une archive et un SHA256

Linux / macOS

```
mkdir -p dist/releases

                                cd build/nuitka
                                zip -r ../../dist/releases/app-linux-x86_64-py312-standalone.zip app.dist
                                cd ../..

                                sha256sum dist/releases/*.zip > dist/releases/SHA256SUMS.txt
```

Windows PowerShell

```
New-Item -ItemType Directory -Force dist\releases

                                Compress-Archive `
                                -Path build\nuitka\app.dist `
                                -DestinationPath dist\releases\app-windows-amd64-py312-standalone.zip `
                                -Force

                                Get-FileHash dist\releases\*.zip -Algorithm SHA256 |
                                Format-Table -AutoSize |
                                Out-File dist\releases\SHA256SUMS.txt
```

> **Bonne pratique :** l'artefact doit toujours être accompagné de son log, son rapport Nuitka, son hash et son rapport de validation.

Release process : de la compilation à la livraison

La release ne doit pas être seulement un build réussi. Elle doit prouver que l'artefact est testable, installable, identifiable et cohérent avec une version du code source.

Étape	Action	Sortie attendue	GO / NO-GO
1. Clean	Nettoyer outputs précédents	Dossier build propre	GO si aucun artefact ancien
2. Install	Installer requirements build	Nuitka + pytest + deps	GO si install OK
3. Tests source	Run pytest / manage.py check	Tests verts	NO-GO si test rouge
4. Build	Nuitka standalone/module/onefile	Artefact généré	NO-GO si build échoue
5. Runtime test	Lancer binaire ou importer module	Smoke test OK	NO-GO si crash
6. Package	Zip / wheel / archive	Artefact final	GO si contenu propre
7. Hash	Calcul SHA256	SHA256SUMS.txt	GO si généré
8. Release notes	Documenter contenu	RELEASE_NOTES.md	GO si clair
9. Upload	GitHub release / artefact store	Release disponible	GO si artefacts attachés

---

Template release notes

```
# Release 0.1.0

                                ## Artefacts
                                - Windows standalone Python 3.12
                                - Linux standalone Python 3.12

                                ## Build environment
                                - Python:
                                - Nuitka:
                                - OS:
                                - Compiler:

                                ## Validation
                                - Source tests: OK
                                - Nuitka build: OK
                                - Binary smoke test: OK
                                - Packaging inspection: OK

                                ## Known limitations
                                - Onefile not provided in this release.
                                - Build must match target OS and Python ABI.

                                ## SHA256
                                See SHA256SUMS.txt
```

Tag Git recommandé

```
# Créer un tag versionné
                                git tag -a v0.1.0 -m "Release v0.1.0 - Nuitka standalone build"

                                # Pousser le tag
                                git push origin v0.1.0
```

> **Conseil :** déclencher les releases CI sur tag `v*`. Cela évite de produire des livrables officiels à chaque commit.

Checklist build industriel Nuitka

#	Contrôle	GO si...	NO-GO si...
1	Script de build versionné	Makefile, PowerShell ou Python script présent	Commande manuelle non documentée
2	Requirements build	`requirements-build.txt` présent	Dépendances installées à la main
3	Tests source	`pytest` ou checks OK	Compilation lancée malgré tests rouges
4	Logs build	`build/logs/build.log` généré	Aucune trace du build
5	Report Nuitka	`compilation-report.xml` généré	Aucun rapport
6	Validation binaire	Smoke test runtime OK	Artefact non exécuté
7	Packaging propre	Archive / wheel inspectée	Sources sensibles ou secrets inclus
8	Nommage artefact	Version + OS + Python + mode visibles	Fichier générique `app.zip`
9	Checksum	SHA256 disponible	Aucune vérification d'intégrité
10	CI/CD	Build reproductible hors machine dev	Fonctionne seulement en local

---

Définition d'un build “propre”

```
Build propre
                                │
                                ├── script versionné
                                ├── environnement documenté
                                ├── tests exécutés
                                ├── Nuitka report généré
                                ├── logs conservés
                                ├── artefact testé
                                ├── archive nommée
                                ├── SHA256 calculé
                                └── release notes écrites
```

Pour IDEO-Lab

Approche recommandée : créer un script Python unique `scripts/build_nuitka.py`, puis l'appeler depuis Makefile, PowerShell et GitHub Actions. Comme cela, la logique de build reste centralisée.

Étape suivante : ajouter un script de validation qui lance réellement l'artefact produit, puis génère un rapport `validation-report.md`.

Commande finale de référence

```
python scripts/build_nuitka.py \
                        --app app.py \
                        --mode standalone \
                        --jobs 8
```

6.2 Dépannage Nuitka — imports, antivirus, Python, MSVC/MinGW, chemins, DLL, performance

Méthode générale : classer l'erreur par couche

Le dépannage Nuitka devient beaucoup plus simple si l'on classe d'abord le problème. Une erreur peut venir du code Python, de Nuitka, du compilateur C/C++, des imports dynamiques, des fichiers data, des dépendances natives, du mode onefile, de l'antivirus ou de la compatibilité entre l'OS, Python et l'architecture.

Arbre de diagnostic global

```
Problème Nuitka
                                │
                                ├── Le script source échoue ?
                                │   └── Corriger le code Python avant Nuitka
                                │
                                ├── Nuitka ne démarre pas ?
                                │   ├── Vérifier venv
                                │   ├── Vérifier python -m nuitka --version
                                │   └── Réinstaller Nuitka dans le bon Python
                                │
                                ├── Le build échoue ?
                                │   ├── Compilateur absent
                                │   ├── Option incorrecte
                                │   ├── Plugin manquant
                                │   └── Version Python / toolchain incompatible
                                │
                                ├── Le binaire démarre puis crash ?
                                │   ├── Import dynamique absent
                                │   ├── Fichier data absent
                                │   ├── DLL / SO manquante
                                │   └── Chemin runtime fragile
                                │
                                ├── Standalone OK mais onefile KO ?
                                │   ├── Extraction temporaire
                                │   ├── Chemin relatif
                                │   └── Antivirus / politique sécurité
                                │
                                └── Performance mauvaise ?
                                ├── Build trop lourd
                                ├── Onefile lent au démarrage
                                ├── Pas de cache compilateur
                                └── Mauvais périmètre compilé
```

Tableau synthèse des erreurs fréquentes

Symptôme	Cause fréquente	Correctif
Build ne trouve pas de compilateur	MSVC / GCC absent	Installer Build Tools ou `build-essential`
`ImportError` au runtime	Import dynamique invisible	`--include-module` ou `--include-package`
Fichier config absent	Data file non inclus	`--include-data-files`
Template introuvable	Dossier templates non livré	`--include-data-dir` ou packaging Django
Antivirus bloque l'exe	Faux positif Windows	Signer, tester standalone, éviter onefile au début
Onefile échoue, standalone marche	Extraction temporaire ou chemins	Déboguer en standalone
Python 3.13 + MinGW échoue	Toolchain / support moins confortable	Tester MSVC / ClangCL ou Python 3.12

Règle d'or : revenir au dernier niveau qui fonctionne : source Python, build simple, standalone, puis onefile.

Problèmes de compilateur C/C++

Nuitka génère du code C/C++, puis délègue au compilateur système. Si la toolchain native est absente ou incompatible, le build échoue avant même d'arriver aux vrais problèmes applicatifs.

Commandes de vérification

```
# Commun
python -m nuitka --version
python -c "import sys; print(sys.version); print(sys.executable)"

# Linux
gcc --version
g++ --version
which gcc
which g++

# Windows
where python
where pip

# Windows : vérifier depuis Developer Command Prompt si nécessaire
cl
```

Installation Linux / Ubuntu

```
sudo apt update
sudo apt install -y build-essential ccache patchelf

gcc --version
g++ --version
```

Correctifs par plateforme

Plateforme	Cause probable	Solution
Windows	MSVC absent	Installer Visual Studio Build Tools avec Desktop development with C++.
Windows	Mauvais terminal	Tester dans Developer Command Prompt ou vérifier PATH.
Windows	MinGW incompatible / incomplet	Préférer MSVC pour un premier build sérieux.
Linux	`gcc` absent	`sudo apt install build-essential`
Linux	`patchelf` absent	`sudo apt install patchelf`
macOS	Clang absent	Installer Xcode Command Line Tools.

> **Conseil Windows :** pour démarrer proprement, utiliser Python 3.11/3.12 64-bit avec MSVC Build Tools est souvent plus confortable qu'un couple Python très récent + MinGW.

Signatures d'erreurs typiques

Message / symptôme	Diagnostic	Action
`gcc: command not found`	GCC absent	Installer `build-essential`.
`cl is not recognized`	MSVC absent ou PATH non chargé	Installer Build Tools ou ouvrir Developer Prompt.
Erreur de link	Librairie native manquante	Lire log, vérifier SDK / dépendances.
Build échoue uniquement en CI	Runner incomplet	Installer packages système dans workflow.

Imports manquants et modules dynamiques

Les imports dynamiques sont l'une des causes les plus fréquentes d'erreurs runtime. Le code fonctionne en Python source, mais le binaire échoue parce que Nuitka n'a pas pu deviner qu'un module serait chargé par son nom texte.

Exemple fragile

```
import importlib

                                plugin_name = "myapp.plugins.csv_exporter"
                                plugin = importlib.import_module(plugin_name)
                                plugin.run()
```

Correctif Nuitka

```
python -m nuitka \
                                --mode=standalone \
                                --include-module=myapp.plugins.csv_exporter \
                                --include-module=myapp.plugins.html_exporter \
                                app.py
```

Diagnostic imports

Pattern	Risque	Correctif
`import package.module`	Faible	Normalement détecté.
`importlib.import_module(name)`	Fort	`--include-module`
Plugins déclarés en YAML	Fort	`--include-package`
Django dotted path setting	Moyen à fort	Inclure le module backend.
Package optionnel	Moyen	Tester le chemin fonctionnel réel.

Règle : si le nom du module est une chaîne de caractères, partez du principe que Nuitka peut avoir besoin d'une inclusion explicite.

Script d'auto-test des imports dynamiques

```
from __future__ import annotations

                        import importlib


                        DYNAMIC_MODULES = [
                        "myapp.plugins.csv_exporter",
                        "myapp.plugins.html_exporter",
                        "myapp.rules.default_rules",
                        ]


                        def main() -> int:
                        for module_name in DYNAMIC_MODULES:
                        module = importlib.import_module(module_name)
                        print("OK import:", module_name, module)

                        return 0


                        if __name__ == "__main__":
                        raise SystemExit(main())
```

Bonne pratique : intégrer ce test dans un mode `--self-check` ou dans pytest, puis le lancer sur le binaire standalone.

Fichiers absents, templates introuvables, chemins fragiles

Une application compilée échoue souvent car elle lit un fichier qui existait dans le repo source, mais qui n'a pas été inclus dans le livrable. Cela concerne les fichiers JSON, YAML, templates, static, certificats, règles métier et fichiers de configuration.

Options correctives

```
# Inclure un fichier de config
                                python -m nuitka \
                                --mode=standalone \
                                --include-data-files=config/default.yaml=config/default.yaml \
                                app.py

                                # Inclure des règles JSON
                                python -m nuitka \
                                --mode=standalone \
                                --include-data-files=rules/*.json=rules/ \
                                app.py

                                # Inclure dossiers templates et static
                                python -m nuitka \
                                --mode=standalone \
                                --include-data-dir=templates=templates \
                                --include-data-dir=static=static \
                                app.py
```

Matrice symptômes / solutions

Erreur	Cause	Solution
`FileNotFoundError`	Fichier non inclus	`--include-data-files`
`TemplateDoesNotExist`	Templates absents	`--include-data-dir=templates=templates`

Erreur	Cause	Solution
CSS / JS manquant	Static absent	Inclure static ou corriger packaging Django.
Fonctionne depuis repo seulement	Chemin relatif fragile	Tester hors dossier source.
Onefile ne trouve pas data	Chemin incompatible extraction	Corriger base runtime ou rester standalone.

Mauvais pattern de chemin

```
from pathlib import Path

                        # Fragile : dépend du dossier courant
                        config_path = Path("config/default.yaml")
                        content = config_path.read_text(encoding="utf-8")
```

Meilleur pattern

```
from pathlib import Path
                        import sys


                        def runtime_base_dir() -> Path:
                        if getattr(sys, "frozen", False):
                        return Path(sys.executable).resolve().parent

                        return Path(__file__).resolve().parent


                        def read_config() -> str:
                        base_dir = runtime_base_dir()
                        config_path = base_dir / "config" / "default.yaml"
                        return config_path.read_text(encoding="utf-8")
```

Test obligatoire : copier le dossier `.dist` dans `/tmp` ou `C:\temp`, puis lancer l'application depuis là-bas.

DLL, SO, DYLIB : dépendances natives manquantes

Certains packages Python dépendent de bibliothèques natives. Quand elles manquent, le binaire peut compiler correctement mais échouer au lancement avec une erreur de chargement.

Commandes d'inspection

```
# Linux : voir les dépendances natives
                        ldd build/nuitka/app.dist/app

                        # Linux : chercher les .so
                        find build/nuitka/app.dist -name "*.so*" -type f

                        # Windows PowerShell : lister les DLL
                        Get-ChildItem -Recurse build\nuitka\app.dist -Filter *.dll

                        # macOS : dépendances
                        otool -L build/nuitka/app.dist/app
```

Erreurs typiques

Erreur	Plateforme	Diagnostic
`DLL load failed`	Windows	DLL absente ou incompatible.
`cannot open shared object file`	Linux	Librairie `.so` absente.
`image not found`	macOS	`.dylib` introuvable.
Crash au démarrage	Toutes	Dépendance native incompatible.

> **Conseil :** tester sur une machine cible propre est plus fiable que tester uniquement sur la machine développeur, qui contient souvent déjà les DLL/SO nécessaires.

Packages souvent concernés

Famille	Exemples	Risque
Base de données	PostgreSQL, MySQL, MariaDB clients	Client natif absent.
Crypto / TLS	OpenSSL, cryptography	DLL ou SO incompatible.
Parsing XML	lxml, libxml2, libxslt	Librairie native absente.
Scientifique	numpy, scipy, pandas	Dépendances BLAS / natives.
GUI	Qt, Tk, GTK	Plugins et libs graphiques.

Onefile : erreurs spécifiques et méthode de debug

Le mode onefile est pratique pour livrer un seul exécutable, mais c'est le mode le moins simple à déboguer. Il peut échouer à cause de l'extraction temporaire, de chemins relatifs, de fichiers data ou de blocages antivirus.

Cycle onefile

```
onefile.exe / onefile.bin
                                │
                                ├── Start
                                ├── Create / use temporary extraction area
                                ├── Extract internal payload
                                ├── Launch compiled program
                                ├── Program reads files / imports modules
                                └── Cleanup / exit
```

Commande de build

```
python -m nuitka \
                                --mode=onefile \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/onefile-report.xml \
                                --assume-yes-for-downloads \
                                app.py
```

Problèmes fréquents onefile

Symptôme	Cause probable	Réponse
Standalone OK, onefile KO	Chemin runtime différent	Corriger accès fichiers.
Démarrage lent	Extraction payload	Mesurer, accepter ou rester standalone.
Antivirus bloque	Fichier auto-extractible inconnu	Signer ou livrer standalone.
Fichier data absent	Inclusion incorrecte	Tester en standalone d'abord.

Erreur classique : commencer directement par onefile. Toujours valider standalone avant onefile.

Méthode de debug

```
Onefile error
                        │
                        ├── Rebuild in standalone
                        │   └── python -m nuitka --mode=standalone app.py
                        │
                        ├── Run from .dist folder
                        │   └── detect missing files / DLL / imports
                        │
                        ├── Fix standalone first
                        │
                        ├── Rebuild onefile
                        │
                        └── If onefile still fails
                        ├── inspect runtime paths
                        ├── test antivirus / security policy
                        └── consider shipping standalone
```

Windows : antivirus, SmartScreen, PowerShell, chemins

Sous Windows, les problèmes les plus courants sont liés à l'antivirus, au mode onefile, à PowerShell, aux chemins longs, à la mauvaise version de Python, ou à une toolchain MSVC incomplète.

Checklist Windows

Contrôle	Commande / action
Python correct	`where python`
Venv actif	`python -c "import sys; print(sys.executable)"`
Nuitka installé	`python -m nuitka --version`
Build Tools installés	Visual Studio Installer.
PowerShell autorise venv	`Set-ExecutionPolicy RemoteSigned -Scope CurrentUser`
Chemin court	Utiliser `C:\dev\project`.

Antivirus / SmartScreen

Symptôme	Cause	Solution
Exe supprimé après build	Antivirus	Tester standalone, exclure dossier build en dev.

Symptôme	Cause	Solution
Alerte SmartScreen	Binaire inconnu non signé	Signer le binaire pour distribution.
Onefile bloqué	Auto-extraction suspecte	Livrer standalone ou signer.
Build très lent	Antivirus scanne les fichiers générés	Exclusion temporaire du dossier build.

Pour une distribution professionnelle : prévoir signature de code, nommage clair, release notes et checksum SHA256.

Commandes utiles Windows

```
# Voir Python utilisé
                        where python
                        python --version
                        python -c "import sys; print(sys.executable)"

                        # Vérifier Nuitka
                        python -m nuitka --version

                        # Nettoyer build
                        Remove-Item -Recurse -Force build\nuitka, build\logs, build\reports -ErrorAction
SilentlyContinue

                        # Lancer un build standalone
                        python -m nuitka `
                        --mode=standalone `
                        --output-dir=build\nuitka `
                        --remove-output `
                        --assume-yes-for-downloads `
                        app.py
```

Versions Python, ABI, architecture et compatibilité

Les modules compilés Nuitka sont liés à la plateforme, à l'architecture et à la version Python. Un module `.pyd` ou `.so` compilé pour Python 3.12 ne doit pas être supposé compatible avec Python 3.11 ou 3.13.

Vérifier environnement

```
python -c "import sys; print(sys.version)"
                                python -c "import sys; print(sys.executable)"
                                python -c "import platform; print(platform.platform());
print(platform.machine())"
                                python -m nuitka --version
```

Nom typique d'extension

```
Windows:
                                scoring.cp312-win_amd64.pyd

                                Linux:
                                scoring.cpython-312-x86_64-linux-gnu.so

                                macOS:
                                scoring.cpython-312-darwin.so
```

Risques fréquents

Problème	Cause	Solution
Module compilé ne s'importe pas	Mauvaise version Python	Recompiler pour la version cible.
Extension Windows livrée sur Linux	Artefact mauvais OS	Builder par OS.
Erreur architecture	x86_64 vs ARM64	Builder sur architecture cible.
Python 3.13 plus difficile	Écosystème / toolchain encore récent	Tester Python 3.12 si blocage.

Règle release : nommer les artefacts avec OS, architecture et version Python. Exemple : `addon-0.1.0-windows-amd64-py312.zip`.

Performance : build lent, binaire lent, onefile lent

Il faut distinguer trois sujets : le temps de compilation, le temps de démarrage, et la performance métier du programme. Nuitka peut améliorer certains cas, mais ce n'est pas un accélérateur universel.

Optimiser le temps de build

```
# Utiliser plusieurs jobs
                                python -m nuitka \
                                --mode=standalone \
                                --jobs=8 \
                                --show-progress \
                                app.py

                                # Linux : installer ccache
                                sudo apt install -y ccache
                                ccache --show-stats

                                # Observer mémoire
                                python -m nuitka \
                                --mode=standalone \
                                --show-memory \
                                app.py
```

Mesurer runtime

```
# Linux / macOS
                                /usr/bin/time -v python app.py
                                /usr/bin/time -v ./build/nuitka/app.dist/app

                                # Windows PowerShell
                                Measure-Command { python .\app.py }
                                Measure-Command { .\build\nuitka\app.dist\app.exe }
```

Attention : onefile peut être plus lent au démarrage à cause de l'extraction. Ce n'est pas forcément une régression du code métier.

Tableau performance

Symptôme	Cause probable	Correction
Build très long	Projet volumineux, pas de cache	`--jobs`, `ccache`, périmètre plus petit.
Mémoire élevée au build	Compilation large	Réduire jobs, compiler modules ciblés.
Onefile démarre lentement	Extraction temporaire	Utiliser standalone pour usage fréquent.
Binaire pas plus rapide	Code I/O ou libs C déjà optimisées	Mesurer avant/après, ne pas promettre de gain.
Build CI lent	Pas de cache pip / compilateur	Ajouter cache CI et limiter matrix.

Conseil : pour protéger un addon Django, compiler un petit module métier est souvent plus efficace que compiler une grosse application complète.

Checklist de dépannage rapide

#	Contrôle	Commande / action	GO si...
1	Source Python OK	`python app.py`	Pas de traceback.
2	Tests OK	`python -m pytest`	Tests verts.
3	Bon venv	`python -c "import sys; print(sys.executable)"`	Chemin pointe vers `.venv`.
4	Nuitka visible	`python -m nuitka --version`	Version affichée.
5	Compilateur présent	`gcc --version` ou MSVC	Toolchain détectée.
6	Standalone testé	`--mode=standalone`	Dossier `.dist` fonctionne.
7	Imports dynamiques inclus	`--include-module` / `--include-package`	Aucun `ModuleNotFoundError`.
8	Data files inclus	`--include-data-files`	Aucun `FileNotFoundError`.
9	DLL / SO présentes	Inspecter `.dist`	Aucune erreur native.
10	Test hors repo	Copier dans `/tmp` ou `C:\temp`	L'application démarre.

Commande diagnostic complète

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --show-memory \
                                --report=build/reports/troubleshooting-report.xml \
                                --assume-yes-for-downloads \
                                --include-package=my_package \
                                app.py
```

Rapport à conserver

```
build/
                                     │
                                     ├── logs/
                                     │   └── build.log
                                     │
                                     ├── reports/
                                     │   ├── troubleshooting-report.xml
                                     │   ├── environment.txt
                                     │   └── validation-report.md
                                     │
                                     └── nuitka/
                                     └── app.dist/
```

Conclusion : ne jamais déboguer directement en onefile. Corriger d'abord le source, puis le build simple, puis le standalone, et seulement ensuite le onefile.

7.1 Plan pas-à-pas Nuitka — valider progressivement que la compilation est jouable

Objectif : vérifier Nuitka sans brûler les étapes

L'objectif n'est pas de compiler immédiatement tout un projet Django ou tout un addon. La bonne approche consiste à valider Nuitka par paliers : d'abord l'environnement, ensuite un script minimal, puis un script plus réaliste, puis un standalone, puis éventuellement un onefile, puis un vrai module métier de l'addon, et enfin le packaging.

Progression recommandée

```
Nuitka validation path
                                │
                                ├── Step 1 : Environment
                                │   └── Python, venv, pip, Nuitka, compiler
                                │
                                ├── Step 2 : Hello build
                                │   └── Minimal script compiled and executed
                                │
                                ├── Step 3 : Realistic script
                                │   └── Imports, functions, exit code, stdout
                                │
                                ├── Step 4 : Standalone
                                │   └── .dist folder tested outside source tree
                                │
                                ├── Step 5 : Onefile
                                │   └── Single executable after standalone success
                                │
                                ├── Step 6 : Django addon core
                                │   └── Compile protected_core/scoring.py
                                │
                                └── Step 7 : Package release
                                └── Installable wheel / archive with compiled core
```

Principe : chaque étape doit produire une preuve. Pas de preuve, pas d'étape suivante.

Pourquoi cette méthode est saine ?

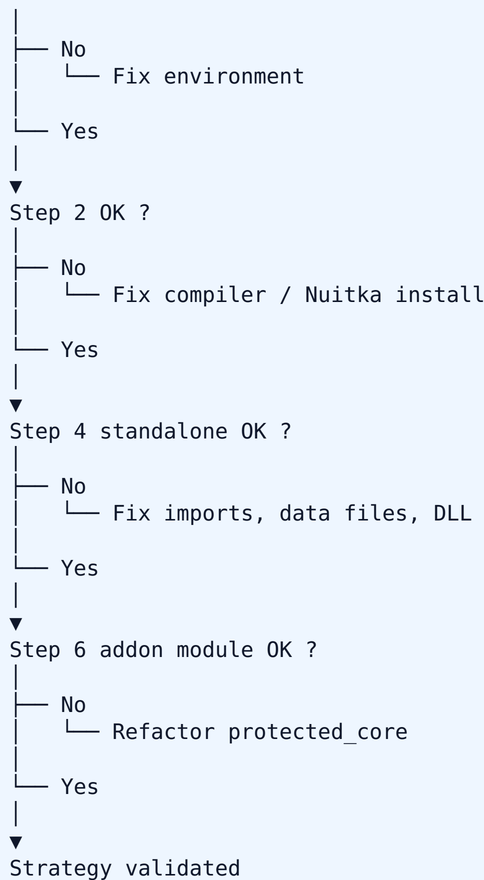
Approche	Résultat
Compiler tout tout de suite	Erreurs difficiles à isoler.
Commencer par onefile	Debug plus compliqué.
Compiler un module instable	Rebuild permanent, frustration.
Avancer par paliers	Diagnostic clair et décisions Go / No-Go.
Tester chaque artefact	Validation réaliste de la stratégie.

Point clé : si les étapes 1 à 4 passent, Nuitka est déjà jouable techniquement. Si l'étape 6 passe sur l'addon, la stratégie de protection partielle devient réaliste.

Roadmap complète en 7 étapes

Étape	Objectif	Action	Critère GO	NO-GO si...
1	Installer Nuitka dans un venv propre	Créer `.venv`, installer Nuitka, vérifier compilateur.	`python -m nuitka --version` OK.	Nuitka ou compilateur introuvable.
2	Compiler `hello_nuitka.py`	Build minimal.	Binaire généré et exécutable.	Erreur de build ou binaire absent.
3	Compiler un script réaliste	Imports standards, fonctions, code retour, stdout.	Même résultat source / binaire.	Résultat différent ou exit code différent.
4	Compiler en standalone	Produire un dossier `.dist`.	Dossier autonome testé hors repo.	Fichier, DLL ou import manquant.
5	Compiler en onefile	Produire un exécutable unique.	Onefile fonctionne comme standalone.	Crash extraction, antivirus, chemins.
6	Compiler un module critique de l'addon	`protected_core/scoring.py` en `.pyd` / `.so`.	Import OK dans Django.	ImportError, ABI incompatible, fallback non maîtrisé.
7	Packager l'addon	Wheel / archive avec module compilé, templates, static.	Installation pip locale OK.	Sources sensibles livrées ou package incomplet.

Diagramme décisionnel



```
Step 1 OK ?
                        │
                        ├── No
                        │   └── Fix environment
                        │
                        └── Yes
                        │
                        ▼
                        Step 2 OK ?
                        │
                        ├── No
                        │   └── Fix compiler / Nuitka install
                        │
                        └── Yes
                        │
                        ▼
                        Step 4 standalone OK ?
                        │
                        ├── No
                        │   └── Fix imports, data files, DLL
                        │
                        └── Yes
                        │
                        ▼
                        Step 6 addon module OK ?
                        │
                        ├── No
                        │   └── Refactor protected_core
                        │
                        └── Yes
                        │
                        ▼
                        Strategy validated
```

Décision centrale : le vrai jalon stratégique est l'étape 6. C'est là que l'on prouve que Nuitka protège réellement une partie utile de l'addon.

Step 1-2 : environnement propre + premier hello build

Ces deux premières étapes valident la fondation : Python, venv, pip, Nuitka, dépendances recommandées et compilateur C/C++. Tant que cela ne fonctionne pas, il est inutile de tester un vrai projet.

Step 1 — environnement

```
# Create virtual environment
python -m venv .venv

# Windows CMD
.venv\Scripts\activate.bat

# Linux / macOS
source .venv/bin/activate

# Upgrade packaging tools
python -m pip install --upgrade pip setuptools wheel

# Install Nuitka
python -m pip install -U nuitka ordered-set zstandard

# Verify
python -m nuitka --version
```

Vérifications système

```
# Linux
gcc --version
g++ --version

# Windows
where python
python --version
python -m nuitka --version
```

Step 2 — hello build

```
# hello_nuitka.py
from __future__ import annotations

import sys


def main() -> int:
print("hello from Nuitka")
print(f"python={sys.version.split()[0]}")
return 0


if __name__ == "__main__":
raise SystemExit(main())
```

```
# Run source
python hello_nuitka.py

# Compile
python -m nuitka hello_nuitka.py

# Run binary
# Windows
hello_nuitka.exe

# Linux / macOS
./hello_nuitka
```

GO Step 2 : le script source fonctionne, le build réussit, le binaire s'exécute et retourne un code `0`.

Causes NO-GO fréquentes

Symptôme	Cause probable	Correction
`No module named nuitka`	Nuitka installé dans un autre Python	Utiliser `python -m pip install -U nuitka` dans le venv actif.
Compilateur absent	MSVC / GCC non installé	Installer Build Tools ou `build-essential`.
Binaire absent	Build échoué	Lire le log complet de Nuitka.

Step 3-4 : script réaliste + standalone

Une fois le hello build validé, il faut tester un script un peu plus représentatif : imports standards, fonctions, code retour, sortie déterministe. Ensuite, le mode standalone valide le vrai packaging.

Step 3 — script réaliste

```
from __future__ import annotations

                                import platform
                                import sys
                                from pathlib import Path


                                def compute_score(values: list[int]) -> int:
                                return sum(value * 3 for value in values) + 17


                                def main() -> int:
                                print("Nuitka realistic smoke test")
                                print(f"python={sys.version.split()[0]}")
                                print(f"platform={platform.system()} {platform.machine()}")
                                print(f"cwd={Path.cwd()}")
                                print(f"score={compute_score([1, 2, 3, 4])}")
                                return 0


                                if __name__ == "__main__":
                                raise SystemExit(main())
```

```
python realistic_test.py
                                python -m nuitka realistic_test.py
                                ./realistic_test
```

Step 4 — standalone

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/standalone-report.xml \
                                --assume-yes-for-downloads \
                                realistic_test.py
```

Tester hors dossier source

```
# Linux / macOS
                                mkdir -p /tmp/nuitka_standalone_test
                                cp -r build/nuitka/realistic_test.dist /tmp/nuitka_standalone_test/
                                cd /tmp/nuitka_standalone_test/realistic_test.dist
                                ./realistic_test

                                # Windows PowerShell
                                New-Item -ItemType Directory -Force C:\temp\nuitka_standalone_test
                                Copy-Item -Recurse build\nuitka\realistic_test.dist C:
\temp\nuitka_standalone_test\
                                cd C:\temp\nuitka_standalone_test\realistic_test.dist
                                .\realistic_test.exe
```

Critères de validation Step 3-4

Contrôle	GO	NO-GO
Sortie source/binaire	Résultat métier identique	Résultat différent
Code retour	`0`	Différent de `0`
Dossier `.dist`	Présent et exécutable	Manquant ou incomplet
Test hors repo	Fonctionne	Dépend du dossier source

GO important : si le standalone fonctionne hors du repo, Nuitka est techniquement viable pour des livrables simples.

Step 5 : onefile, uniquement après standalone

Le mode onefile est attractif parce qu'il produit un seul fichier. Mais il est plus difficile à diagnostiquer. Il doit donc être considéré comme une étape de confort de distribution, pas comme une étape de diagnostic.

Commande onefile

```
python -m nuitka \
                                --mode=onefile \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/onefile-report.xml \
                                --assume-yes-for-downloads \
                                realistic_test.py
```

Tester onefile

```
# Windows
                                build\nuitka\realistic_test.exe

                                # Linux / macOS
                                ./build/nuitka/realistic_test.bin
```

Décision onefile

Situation	Décision
Standalone OK, onefile OK	Onefile utilisable pour livraison simple.
Standalone OK, onefile KO	Garder standalone pour l'instant.
Onefile bloqué par antivirus	Signer ou livrer standalone.
Onefile lent au démarrage	Mesurer ; standalone peut être meilleur.
Application avec beaucoup de fichiers data	Standalone souvent plus lisible.

Ne pas bloquer le projet sur onefile : si standalone fonctionne, la stratégie Nuitka reste valable même si onefile demande plus de travail.

Pièges onefile

```
Onefile problems
                        │
                        ├── Temporary extraction path
                        ├── Relative file paths
                        ├── Antivirus false positives
                        ├── Slower startup
                        ├── Data files harder to inspect
                        └── Harder debugging
```

Step 6 : compiler un vrai module critique de l'addon Django

Cette étape est le vrai test stratégique. Il ne s'agit plus de compiler un exemple, mais un module réel de `ideolab_admin_tools`. Le meilleur premier candidat est un module court, stable, testable, à valeur métier claire : par exemple `protected_core/scoring.py`.

Structure cible

```
ideolab_admin_tools/
                                │
                                ├── admin.py
                                ├── views.py
                                ├── apps.py
                                ├── core_api.py
                                │
                                └── protected_core/
                                ├── __init__.py
                                ├── scoring.py        -> source dev
                                ├── scoring_py.py     -> fallback dev optionnel
                                └── scoring.pyd/.so   -> module compilé release
```

Commande module

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/scoring-module-report.xml \
                                ideolab_admin_tools/protected_core/scoring.py
```

Tests Django après compilation

```
# Copy compiled module into package
                                cp build/nuitka_modules/scoring*.so ideolab_admin_tools/protected_core/

                                # Direct import
                                python -c "import ideolab_admin_tools.protected_core.scoring;
print('compiled import OK')"

                                # Django checks
                                python manage.py check

                                # App tests
                                python manage.py test ideolab_admin_tools
```

Validation	GO si...
Import direct	Le module compilé s'importe.
API wrapper	`core_api.py` appelle le module compilé.
Django check	`manage.py check` OK.
Admin	Pages custom admin accessibles.
Fallback	Désactivé en mode prod.

> **GO stratégique :** si `scoring.py` compilé s'importe dans Django et fonctionne via `core_api.py`, alors la protection partielle de l'addon est réaliste.

Step 7 : packager l'addon avec le cœur compilé

La dernière étape consiste à vérifier que l'addon reste installable et utilisable une fois le module compilé intégré. Il faut inclure les extensions `.pyd` ou `.so`, conserver templates/static, exclure les sources sensibles si la stratégie est fermée, puis tester dans un venv propre.

MANIFEST.in recommandé

```
recursive-include ideolab_admin_tools/templates *
                                recursive-include ideolab_admin_tools/static *
                                recursive-include ideolab_admin_tools/protected_core *.pyd
                                recursive-include ideolab_admin_tools/protected_core *.so
                                recursive-include ideolab_admin_tools/protected_core/rules *.json

                                recursive-exclude tests *
                                recursive-exclude docs/internal *
                                recursive-exclude ideolab_admin_tools/protected_core *_scratch.py
                                recursive-exclude ideolab_admin_tools/protected_core *_dev.py
```

Build wheel

```
python -m pip install build
                                python -m build
```

Test dans un venv propre

```
python -m venv /tmp/ideolab_addon_test
                                source /tmp/ideolab_addon_test/bin/activate

                                python -m pip install --upgrade pip
                                python -m pip install dist/ideolab_admin_tools-*.whl

                                python -c "import ideolab_admin_tools; print('addon import OK')"
                                python -c "import ideolab_admin_tools.protected_core.scoring;
print('compiled core OK')"
```

Inspection de wheel

```
python -m zipfile -l dist/ideolab_admin_tools-*.whl
```

> **NO-GO :** si la wheel contient les sources sensibles `protected_core/scoring.py` en plus du module compilé, la protection est diminuée.

Critères Step 7

Contrôle	GO	NO-GO
Wheel générée	Fichier `.whl` présent	Build package échoue
Installation pip	Installation dans venv propre OK	Package incomplet
Compiled import	`protected_core.scoring` importable	Extension incompatible
Templates/static	Inclus dans package	Admin cassé
Sources sensibles	Absentes selon stratégie	Sources livrées par erreur

---

Go / No-Go : comment décider objectivement ?

La décision doit être factuelle. Nuitka est “jouable” si les étapes techniques passent sans contorsion excessive, si les tests sont reproductibles, et si le module réel de l'addon peut être compilé puis importé dans Django.

Niveau	Décision	Condition	Suite
GO technique minimal	Nuitka est installable	Étape 1 OK	Continuer vers hello build
GO compilation	Nuitka compile sur la machine	Étape 2 OK	Tester script réaliste
GO packaging	Standalone viable	Étape 4 OK	Tester onefile si utile
GO protection partielle	Addon compatible	Étape 6 OK	Industrialiser build module
GO release	Distribution possible	Étape 7 OK	CI/CD + release versionnée
NO-GO temporaire	Blocage technique isolé	Compiler OK mais packaging KO	Réparer imports/data/DLL
NO-GO stratégique	Périmètre trop couplé	Module addon impossible à isoler	Refactoriser avant Nuitka

---

Go fort

- Venv propre OK.
- Build simple OK.
- Standalone OK hors repo.

- Module réel compilé OK.
- Import Django OK.
- Package installable OK.
- Sources sensibles absentes du livrable final.

No-Go temporaire

- Imports dynamiques non inclus.
- Fichiers data absents.
- DLL/SO manquantes.
- Onefile uniquement cassé.
- Fallback Python mal contrôlé.
- Wheel incomplète.

Décision recommandée : ne pas exiger onefile pour dire que Nuitka est viable. Le vrai critère est : standalone OK + module addon compilé importable dans Django.

Risques du plan et réponses associées

Risque	Moment probable	Impact	Réponse
Compilateur non installé	Step 1-2	Build impossible	Installer MSVC / GCC / Clang.
Python trop récent / toolchain inconfortable	Step 1-2	Erreurs difficiles	Tester Python 3.12 + MSVC sous Windows.
Imports dynamiques	Step 4-6	Runtime KO	`--include-module`, `--include-package`.
Fichiers data absents	Step 4-7	FileNotFoundError	`--include-data-files`, packaging Django.
Module addon trop couplé à Django	Step 6	Compilation difficile	Créer DTO + API wrapper + core isolé.
Extension incompatible OS/Python	Step 6-7	ImportError	Builder par OS et version Python.
Sources sensibles livrées	Step 7	Protection annulée	Inspecter wheel et nettoyer release.
Onefile bloqué antivirus	Step 5	Distribution gênée	Signer ou livrer standalone.

Plan de mitigation

```
Risk mitigation
                         │
                         ├── Keep scope small
                         │   └── compile one module first
                         │
                         ├── Keep API stable
                         │   └── Django calls core_api.py
                         │
                         ├── Keep tests mandatory
                         │   └── pytest + manage.py check
                         │
                         ├── Keep packaging explicit
                         │   └── MANIFEST.in + wheel inspection
                         │
                         ├── Keep platform builds separated
                         │   └── Windows build for Windows, Linux build for Linux
                         │
                         └── Keep release clean
                         └── no secrets, no sensitive sources, no internal docs
```

> **Point de vigilance :** si Step 6 échoue, ne pas forcer Nuitka. Refactoriser d'abord le module pour réduire le couplage à Django.

Plan final exécutable : ordre exact recommandé

Voici l'ordre opérationnel recommandé pour valider Nuitka sur un périmètre réaliste, sans se perdre dans des problèmes de packaging ou de protection trop tôt.

Ordre de travail

```
Phase A - Environment
                                │
                                ├── Create venv
                                ├── Install Nuitka
                                ├── Verify compiler
                                └── Compile hello

                                Phase B - Packaging basics
                                │
                                ├── Compile realistic script
                                ├── Compare source/binary
                                ├── Build standalone
                                └── Test outside repo

                                Phase C - Optional onefile
                                │
                                ├── Build onefile
                                ├── Test antivirus/runtime
                                └── Keep standalone if needed

                                Phase D - Real addon core
                                │
                                ├── Isolate protected_core/scoring.py
                                ├── Add core_api.py
                                ├── Compile module
                                ├── Import in Django
                                └── Run manage.py check

                                Phase E - Release
                                │
                                ├── Include .pyd/.so
                                ├── Include templates/static
                                ├── Remove sensitive sources
                                ├── Build wheel
                                ├── Install in clean venv
                                └── Inspect final package
```

Commandes de référence

```
# Step 1
                                python -m venv .venv
                                source .venv/bin/activate
                                python -m pip install -U pip setuptools wheel
                                python -m pip install -U nuitka ordered-set zstandard
                                python -m nuitka --version

                                # Step 2
                                python -m nuitka hello_nuitka.py

                                # Step 4
                                python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/standalone-report.xml \
                                hello_nuitka.py

                                # Step 6
                                python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                ideolab_admin_tools/protected_core/scoring.py

                                # Step 7
                                python -m build
                                python -m zipfile -l dist/ideolab_admin_tools-*.whl
```

Verdict final attendu

Résultat obtenu	Interprétation	Décision
Étapes 1-4 OK	Nuitka est techniquement jouable.	Continuer vers addon.
Étape 5 KO seulement	Onefile pose problème, mais stratégie viable.	Livrer standalone ou corriger plus tard.
Étape 6 OK	Protection partielle réaliste.	Industrialiser build module.
Étape 7 OK	Distribution possible.	Passer CI/CD et release versionnée.
Étape 6 KO	Module trop couplé ou packaging incorrect.	Refactoriser avant compilation.

Conclusion : la validation Nuitka ne se joue pas sur un seul build. Elle se joue sur une chaîne : environnement OK, standalone OK, module addon OK, packaging OK. C'est cette chaîne qui prouve que Nuitka est réellement utilisable.

7.2 Cheat-sheet Nuitka — commandes prêtes à copier Windows, Linux, standalone, onefile, modules

Installation rapide Nuitka

Cette section regroupe les commandes minimales pour préparer un environnement propre. Le réflexe professionnel est toujours le même : créer un `.venv`, l'activer, mettre `pip` à jour, installer Nuitka, puis vérifier que la commande répond.

Windows CMD

```
python -m venv .venv
                                .venv\Scripts\activate.bat

                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -U nuitka ordered-set zstandard

                                python -m nuitka --version
```

Windows PowerShell

```
python -m venv .venv
                                .venv\Scripts\Activate.ps1

                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -U nuitka ordered-set zstandard

                                python -m nuitka --version
```

Linux / Ubuntu / Debian

```
sudo apt update
                                sudo apt install -y python3 python3-venv python3-pip
                                sudo apt install -y build-essential ccache patchelf

                                python3 -m venv .venv
                                source .venv/bin/activate

                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -U nuitka ordered-set zstandard

                                python -m nuitka --version
```

Contrôle environnement

```
python -c "import sys; print(sys.version); print(sys.executable)"
                                python -m pip --version
                                python -m nuitka --version
```

Vérifications compilateur

Plateforme	Commande	Résultat attendu
Windows	`where python`	Chemin Python correct, idéalement dans le venv.
Windows	`cl`	MSVC visible si Developer Command Prompt utilisé.
Linux	`gcc --version`	GCC installé.
Linux	`g++ --version`	G++ installé.

Plateforme	Commande	Résultat attendu
Linux	`patchelf --version`	Patchelf disponible pour standalone.

Règle simple : toujours lancer Nuitka avec `python -m nuitka`. Cela garantit que l'on utilise le Nuitka installé dans le Python actif.

Commandes de base : accelerated, standalone, onefile

Ces commandes couvrent les trois premiers modes pratiques : build simple pour tester, standalone pour diagnostiquer correctement, onefile pour livrer un fichier unique.

Build simple / accelerated

```
# Syntaxe courte
                                python -m nuitka app.py

                                # Syntaxe explicite
                                python -m nuitka --mode=accelerated app.py

                                # Avec dossier de sortie
                                python -m nuitka \
                                --mode=accelerated \
                                --output-dir=build/nuitka \
                                app.py
```

Standalone

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --assume-yes-for-downloads \
                                app.py
```

Onefile

```
python -m nuitka \
                                --mode=onefile \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --assume-yes-for-downloads \
                                app.py
```

À éviter au début

```
# Trop direct : pas de standalone, pas de rapport, debug difficile
                                python -m nuitka --onefile app.py
```

Ordre conseillé : source Python OK → accelerated OK → standalone OK → onefile seulement ensuite.

Exécuter les artefacts

Mode	Windows	Linux / macOS
Accelerated	`app.exe`	`./app`
Standalone	`build\nuitka\app.dist\app.exe`	`./build/nuitka/app.dist/app`
Onefile	`build\nuitka\app.exe`	`./build/nuitka/app.bin`

Cheat-sheet Windows

Sous Windows, la priorité est de vérifier le bon Python, le bon venv, puis la toolchain MSVC. Pour un premier usage sérieux, Visual Studio Build Tools avec Desktop development with C++ est le chemin le plus stable.

Vérifier Python / venv / Nuitka

```
where python
                                where pip

                                python --version
                                python -m pip --version
                                python -m nuitka --version

                                python -c "import sys; print(sys.executable)"
```

Build standalone Windows

```
python -m nuitka `
                                --mode=standalone `
                                --output-dir=build\nuitka `
                                --remove-output `
                                --show-progress `
                                --assume-yes-for-downloads `
                                app.py
```

Build onefile Windows

```
python -m nuitka `
                                --mode=onefile `
                                --output-dir=build\nuitka `
                                --remove-output `
                                --show-progress `
                                --assume-yes-for-downloads `
                                app.py
```

Nettoyer les builds

```
Remove-Item -Recurse -Force build\nuitka -ErrorAction SilentlyContinue
                                Remove-Item -Recurse -Force build\logs -ErrorAction SilentlyContinue
                                Remove-Item -Recurse -Force build\reports -ErrorAction SilentlyContinue
```

Antivirus : si l'exe est bloqué, tester d'abord en standalone, puis envisager signature de code pour une vraie distribution.

PowerShell : script minimal de build

```
$ErrorActionPreference = "Stop"

                        New-Item -ItemType Directory -Force build\nuitka, build\logs, build\reports | Out-
Null

                        python -m nuitka `
                        --mode=standalone `
                        --output-dir=build\nuitka `
                        --remove-output `
                        --show-progress `
                        --report=build\reports\standalone-report.xml `
                        --assume-yes-for-downloads `
                        app.py *>&1 | Tee-Object build\logs\standalone-build.log

                        if ($LASTEXITCODE -ne 0) {
                        throw "Nuitka build failed with exit code $LASTEXITCODE"
                        }
```

Cheat-sheet Linux / Ubuntu / Debian

Installation système

```
sudo apt update
                                sudo apt install -y python3 python3-venv python3-pip
                                sudo apt install -y build-essential ccache patchelf zip
```

Venv + Nuitka

```
python3 -m venv .venv
                                source .venv/bin/activate

                                python -m pip install --upgrade pip setuptools wheel
                                python -m pip install -U nuitka ordered-set zstandard

                                python -m nuitka --version
```

Build standalone Linux

```
mkdir -p build/logs build/reports

                                python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/standalone-report.xml \
                                --assume-yes-for-downloads \
                                app.py 2>&1 | tee build/logs/standalone-build.log
```

Build onefile Linux

```
mkdir -p build/logs build/reports

                                python -m nuitka \
                                --mode=onefile \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/onefile-report.xml \
                                --assume-yes-for-downloads \
                                app.py 2>&1 | tee build/logs/onefile-build.log
```

Tester hors repo

```
mkdir -p /tmp/nuitka_runtime_test
                                cp -r build/nuitka/app.dist /tmp/nuitka_runtime_test/
                                cd /tmp/nuitka_runtime_test/app.dist
                                ./app
```

Nettoyer

```
rm -rf build/nuitka build/logs build/reports
                                rm -rf *.build *.dist *.onefile-build
```

Inspection Linux

```
# Vérifier dépendances natives
                        ldd build/nuitka/app.dist/app

                        # Chercher les .so
                        find build/nuitka/app.dist -name "*.so*" -type f

                        # Mesurer runtime
                        /usr/bin/time -v ./build/nuitka/app.dist/app
```

Compiler un module Python importable

Le mode module est le plus intéressant pour protéger un cœur métier : scoring, analyse, règles, diagnostic, parsing, moteur propriétaire.

Commande module simple

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                package/core_engine.py
```

Avec rapport

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/core-engine-report.xml \
                                package/core_engine.py
```

Résultats typiques

```
Windows:
                                core_engine.cp312-win_amd64.pyd

                                Linux:
                                core_engine.cpython-312-x86_64-linux-gnu.so

                                macOS:
                                core_engine.cpython-312-darwin.so
```

Tester l'import

```
python -c "import package.core_engine; print('compiled import OK')"
```

Pour un addon Django : compiler d'abord un seul module simple, par exemple `protected_core/scoring.py`.

Copier le module dans le package

Windows

```
copy build\nuitka_modules\core_engine*.pyd package\
```

Linux / macOS

```
cp build/nuitka_modules/core_engine*.so package/
```

Inclure fichiers, templates, static, configs

Les fichiers non Python doivent être inclus explicitement quand ils sont nécessaires au runtime. Cela concerne les templates, static, JSON, YAML, certificats, règles et fichiers de configuration.

Avec data files

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --include-data-files=config.yaml=config.yaml \
                                --include-data-files=rules/*.json=rules/ \
                                app.py
```

Avec dossiers data

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --include-data-dir=templates=templates \
                                --include-data-dir=static=static \
                                app.py
```

Avec imports dynamiques

```
python -m nuitka \
                                --mode=standalone \
                                --include-module=myapp.plugins.csv_exporter \
                                --include-module=myapp.plugins.html_exporter \
                                --include-package=myapp.rules \
                                app.py
```

Exclure du livrable

```
# Ne jamais inclure dans une release fermée :
                                # .env
                                # .git/
                                # tests/
                                # docs/internal/
                                # dev_config.yaml
                                # sources sensibles si remplacées par .pyd/.so
```

Test obligatoire : exécuter le standalone hors du dossier source pour vérifier que les fichiers sont vraiment inclus.

Chemins runtime robustes

```
from pathlib import Path
                        import sys


                        def runtime_base_dir() -> Path:
                        if getattr(sys, "frozen", False):
                        return Path(sys.executable).resolve().parent
                        return Path(__file__).resolve().parent


                        def read_rules() -> str:
                        path = runtime_base_dir() / "rules" / "default_rules.json"
                        return path.read_text(encoding="utf-8")
```

Cheat-sheet spécial Django addon

Pour un addon Django, l'approche recommandée est de garder la couche Django en Python lisible et de compiler uniquement le cœur métier propriétaire.

Compiler le cœur métier

```
python -m nuitka \
                                --mode=module \
                                --output-dir=build/nuitka_modules \
                                --remove-output \
                                --show-progress \
                                --report=build/reports/scoring-report.xml \
                                ideolab_admin_tools/protected_core/scoring.py
```

Copier extension compilée

```
# Windows
                                copy build\nuitka_modules\scoring*.pyd ideolab_admin_tools\protected_core\

                                # Linux / macOS
                                cp build/nuitka_modules/scoring*.so ideolab_admin_tools/protected_core/
```

Tests Django

```
python -c "import ideolab_admin_tools.protected_core.scoring; print('compiled core OK')"

                                python manage.py check

                                python manage.py test ideolab_admin_tools

                                python manage.py runserver
```

MANIFEST.in

```
recursive-include ideolab_admin_tools/templates *
                                recursive-include ideolab_admin_tools/static *
                                recursive-include ideolab_admin_tools/protected_core *.pyd
                                recursive-include ideolab_admin_tools/protected_core *.so
                                recursive-include ideolab_admin_tools/protected_core/rules *.json
                                recursive-exclude tests *
                                recursive-exclude docs/internal *
```

Ne pas compiler au début

Fichier	Pourquoi éviter
`admin.py`	Couche Django Admin, introspection, registre.
`views.py`	HTTP, request, permissions, templates.
`models.py`	ORM, migrations, app registry.
`migrations/`	Historique Django à garder lisible.

Commande clé IDEO-Lab : commencer par `protected_core/scoring.py`, puis élargir vers `analyzer.py` et `rules_engine.py`.

Debug rapide : commandes et réflexes

Diagnostic environnement

```
python -c "import sys; print(sys.version); print(sys.executable)"
                                python -m pip --version
                                python -m nuitka --version

                                # Linux
                                gcc --version
                                g++ --version
                                patchelf --version

                                # Windows
                                where python
                                where pip
```

Build avec rapport

```
python -m nuitka \
                                --mode=standalone \
                                --output-dir=build/nuitka \
                                --remove-output \
                                --show-progress \
                                --show-memory \
                                --report=build/reports/debug-report.xml \
                                --assume-yes-for-downloads \
                                app.py
```

Erreurs fréquentes

Erreur	Correction
`No module named nuitka`	Installer Nuitka dans le venv actif.
`ModuleNotFoundError`	Ajouter `--include-module` ou `--include-package`.
`FileNotFoundError`	Ajouter `--include-data-files`.
`DLL load failed`	Inspecter `.dist`, tester machine cible.
Onefile KO	Revenir en standalone et corriger d'abord là.

Règle de debug : ne jamais déboguer d'abord en onefile. Déboguer en standalone, puis refaire le onefile.

Test runtime automatique

```
python - <<'PY'
                        import subprocess
                        import sys
                        from pathlib import Path

                        if sys.platform == "win32":
                        binary = Path("build/nuitka/app.dist/app.exe")
                        else:
                        binary = Path("build/nuitka/app.dist/app")

                        if not binary.exists():
                        raise SystemExit(f"Missing binary: {binary}")

                        result = subprocess.run([str(binary)], text=True, capture_output=True)
                        print("returncode:", result.returncode)
                        print("stdout:", result.stdout)
                        print("stderr:", result.stderr)

                        if result.returncode != 0:
                        raise SystemExit("Runtime test failed")

                        print("OK - runtime test passed")
                        PY
```

CI / release : commandes utiles

requirements-build.txt

```
nuitka
                                ordered-set
                                zstandard
                                setuptools
                                wheel
                                pytest
                                build
```

Build Python script

```
python scripts/build_nuitka.py \
                                --app app.py \
                                --mode standalone \
                                --jobs 8
```

Tests avant release

```
python -m pytest -v
                                python -m compileall .
                                python scripts/validate_artifact.py
```

Créer archive Linux

```
mkdir -p dist/releases

                                cd build/nuitka
                                zip -r ../../dist/releases/app-linux-x86_64-py312-standalone.zip app.dist
                                cd ../..

                                sha256sum dist/releases/*.zip > dist/releases/SHA256SUMS.txt
```

Créer archive Windows PowerShell

```
New-Item -ItemType Directory -Force dist\releases

                                Compress-Archive `
                                -Path build\nuitka\app.dist `
                                -DestinationPath dist\releases\app-windows-amd64-py312-standalone.zip `
                                -Force

                                Get-FileHash dist\releases\*.zip -Algorithm SHA256 |
                                Format-Table -AutoSize |
                                Out-File dist\releases\SHA256SUMS.txt
```

Git tag release

```
git tag -a v0.1.0 -m "Release v0.1.0 - Nuitka build"
                        git push origin v0.1.0
```

Artefact propre : nommer avec version, OS, architecture, Python et mode. Exemple : `ideolab-admin-tools-0.1.0-windows-amd64-py312-module.zip`.

Super résumé : les commandes à retenir

Top commandes

```
# Install
                                python -m venv .venv
                                source .venv/bin/activate
                                python -m pip install -U pip setuptools wheel
                                python -m pip install -U nuitka ordered-set zstandard

                                # Verify
                                python -m nuitka --version

                                # Simple build
                                python -m nuitka app.py

                                # Standalone
                                python -m nuitka --mode=standalone --assume-yes-for-downloads app.py

                                # Onefile
                                python -m nuitka --mode=onefile --assume-yes-for-downloads app.py

                                # Module
                                python -m nuitka --mode=module package/core_engine.py
```

Top options

```
# Output
                                --output-dir=build/nuitka
                                --remove-output
                                --show-progress
                                --show-memory
                                --report=build/reports/report.xml

                                # Data
                                --include-data-dir=templates=templates
                                --include-data-dir=static=static
                                --include-data-files=config.yaml=config.yaml
                                --include-data-files=rules/*.json=rules/

                                # Imports
                                --include-module=myapp.plugins.csv_exporter
                                --include-package=myapp.plugins

                                # Build comfort
                                --jobs=8
                                --assume-yes-for-downloads
```

Phrase mémo : standalone pour comprendre, onefile pour livrer, module pour protéger un cœur métier.

Commande complète de référence

```
python -m nuitka \
                        --mode=standalone \
                        --output-dir=build/nuitka \
                        --remove-output \
                        --jobs=8 \
                        --show-progress \
                        --show-memory \
                        --report=build/reports/compilation-report.xml \
                        --assume-yes-for-downloads \
                        --include-data-dir=templates=templates \
                        --include-data-dir=static=static \
                        --include-data-files=config.yaml=config.yaml \
                        --include-module=myapp.plugins.csv_exporter \
                        app.py
```

Dernier conseil : ne garde jamais une commande Nuitka longue uniquement dans l'historique du terminal. Mets-la dans un script versionné : `build_nuitka.py`, `build_windows.ps1` ou `Makefile`.

7.3 Ressources Nuitka — documentation officielle, diagnostics, URLs, lectures recommandees

Ressources officielles Nuitka

Cette section regroupe les liens de reference a consulter en priorite. Pour demarrer serieusement, l'ordre conseille est : site officiel, User Manual, tutorial de setup, common issues, puis package configuration si le projet contient des ressources, plugins, imports dynamiques ou dependances particulieres.

Ressource	URL	Usage	Quand consulter ?
Site officiel Nuitka	https://nuitka.net/	Point d'entree general, presentation du compilateur.	Avant tout, pour comprendre le positionnement global.
User Manual	https://nuitka.net/user-documentation/user-manual.html	Reference principale : installation, usage, options, data files, reports.	Des le debut du projet.
User Documentation	https://nuitka.net/user-documentation/	Index general des pages de documentation utilisateur.	Pour naviguer dans les sujets avances.
Tutorial setup and build	https://nuitka.net/user-documentation/tutorial-setup-and-build.html	Tutoriel de demarrage : setup, build, premiers tests.	Pour le premier build Nuitka.
Common issue solutions	https://nuitka.net/user-documentation/common-issue-solutions.html	Solutions aux problemes frequents : standalone, antivirus, Linux, crashs.	Des qu'un build compile mais ne marche pas au runtime.
Package configuration	https://nuitka.net/user-documentation/nuitka-package-config.html	Gestion avancee des packages, data files, dependances speciales.	Pour les projets complexes, frameworks et dependances non triviales.
Changelog	https://nuitka.net/changelog/Changelog.html	Historique des versions, changements, corrections, compatibilites.	Avant upgrade Nuitka ou si un comportement change.

Regle simple : en cas de doute, commencer par le User Manual, puis lire Common Issues avant de multiplier les options au hasard.

Documentation par besoin

Nuitka couvre plusieurs usages : script simple, standalone, onefile, module importable, package complexe, data files, rapport de compilation, performance et distribution. Cette table permet de trouver rapidement la bonne page selon le probleme rencontre.

Besoin	Page	URL	Ce qu'il faut y chercher
Comprendre Nuitka	User Manual	https://nuitka.net/user-documentation/user-manual.html	Requirements, installation, command line usage, modes, reports.
Premier build	Tutorial setup and build	https://nuitka.net/user-documentation/tutorial-setup-and-build.html	Setup initial, test de compilation, workflow de base.
Problemes frequents	Common issue solutions	https://nuitka.net/user-documentation/common-issue-solutions.html	Antivirus Windows, Linux standalone, deployment mode, crashs.
Packages difficiles	Package configuration	https://nuitka.net/user-documentation/nuitka-package-config.html	Configuration de packages, inclusions, dependances speciales.
Suivre evolutions	Changelog	https://nuitka.net/changelog/Changelog.html	Versions, changements de comportement, nouvelles options.

Lecture minimale recommandee

```
Start
                                  │
                                  ├── 1. User Manual
                                  │   └── comprendre les bases et les modes
                                  │
                                  ├── 2. Tutorial setup and build
                                  │   └── faire un premier build
                                  │
                                  ├── 3. Common issue solutions
                                  │   └── connaitre les pieges classiques
                                  │
                                  └── 4. Package configuration
                                  └── gerer packages complexes et data files
```

A consulter avant de poser une issue

- La commande exacte lancee.
- La sortie de `python -m nuitka --version`.
- Le mode utilise : accelerated, standalone, onefile ou module.
- Le systeme cible : Windows, Linux, macOS.
- Le test source Python avant compilation.
- Le log complet du build.
- Le rapport Nuitka si disponible.

Build, options et commandes utiles

Cette section regroupe les liens utiles pour comprendre les options et les workflows de build. Les options essentielles a maitriser sont : `--mode`, `--output-dir`, `--include-data-files`, `--include-data-dir`, `--include-module`, `--include-package`, `--report`, `--jobs`.

Sujet	Ressource	URL	Commandes liees
Modes de compilation	User Manual	https://nuitka.net/user-documentation/user-manual.html	`--mode=standalone`, `--mode=onefile`, `--mode=module`
Standalone	Common Issues	https://nuitka.net/user-documentation/common-issue-solutions.html	`--mode=standalone`
Data files	User Manual	https://nuitka.net/user-documentation/user-manual.html	`--include-data-files`, `--include-data-dir`
Packages complexes	Package configuration	https://nuitka.net/user-documentation/nuitka-package-config.html	`--include-package`, config package
Compilation report	User Manual	https://nuitka.net/user-documentation/user-manual.html	`--report=build/reports/report.xml`

Commandes de reference

```
# Standalone avec rapport
                        python -m nuitka \
                        --mode=standalone \
                        --output-dir=build/nuitka \
                        --remove-output \
                        --show-progress \
                        --report=build/reports/compilation-report.xml \
                        --assume-yes-for-downloads \
                        app.py

                        # Onefile
                        python -m nuitka \
                        --mode=onefile \
                        --output-dir=build/nuitka \
                        --assume-yes-for-downloads \
                        app.py

                        # Module importable
                        python -m nuitka \
                        --mode=module \
                        --output-dir=build/nuitka_modules \
                        package/core_engine.py
```

> **Conseil :** toujours conserver la commande exacte utilisee dans un fichier `build/reports/command.txt` ou dans un script versionne.

Ressources de diagnostic et depannage

Quand un build echoue, il faut eviter de changer dix options a la fois. Le bon reflexe est de classifier l'erreur : source Python, installation Nuitka, compilateur C/C++, import dynamique, data file, DLL/SO, onefile, antivirus ou packaging.

Probleme	Ressource	URL	Action rapide
Antivirus Windows	Common Issues	https://nuitka.net/user-documentation/common-issue-solutions.html	Tester standalone, eviter onefile au debut, envisager signature.
Linux standalone	Common Issues	https://nuitka.net/user-documentation/common-issue-solutions.html	Installer `patchelf`, tester hors repo.
Imports manquants	User Manual	https://nuitka.net/user-documentation/user-manual.html	Ajouter `--include-module` ou `--include-package`.
Packages speciaux	Package configuration	https://nuitka.net/user-documentation/nuitka-package-config.html	Verifier la configuration de package.
Regression apres upgrade	Changelog	https://nuitka.net/changelog/Changelog.html	Comparer versions Nuitka et options.
Issue non resolue	GitHub Issues	https://github.com/Nuitka/Nuitka/issues	Chercher le message exact avant d'ouvrir une issue.

Informations a joindre a un diagnostic

```
python -m nuitka --version
                        python -c "import sys; print(sys.version); print(sys.executable)"
                        python -m pip list

                        # Linux
                        gcc --version
                        g++ --version
                        patchelf --version

                        # Windows
                        where python
                        where pip
```

Debug propre : reproduire le probleme sur un exemple minimal, puis seulement ensuite le reporter au projet complet.

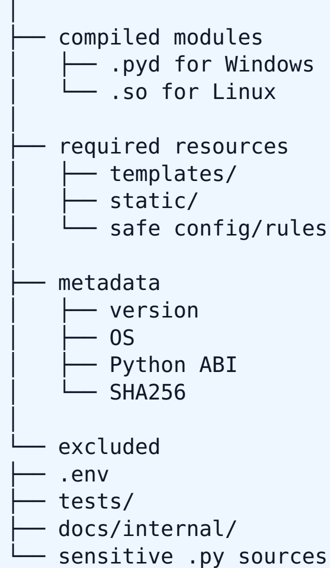
Packaging, wheels, data files et livraison

Pour un projet professionnel, il faut distinguer la compilation Nuitka du packaging Python. Nuitka produit un artefact, mais la distribution peut ensuite prendre plusieurs formes : dossier standalone, onefile, wheel, archive zip, package interne, ou release GitHub.

Ressources packaging

Ressource	URL	Usage
PyPI Nuitka	https://pypi.org/project/Nuitka/	Installation via pip, versions publiees.
Python Packaging User Guide	https://packaging.python.org/	Packaging Python, wheels, pyproject, distribution.
Setuptools package data	https://setuptools.pypa.io/en/latest/userguide/datafiles.html	Inclure templates, static, fichiers package.
Build package	https://pypa-build.readthedocs.io/	Construire wheels et sdists avec `python -m build`.

Checklist packaging



```
Release package
                                 │
                                 ├── compiled modules
                                 │   ├── .pyd for Windows
                                 │   └── .so for Linux
                                 │
                                 ├── required resources
                                 │   ├── templates/
                                 │   ├── static/
                                 │   └── safe config/rules
                                 │
                                 ├── metadata
                                 │   ├── version
                                 │   ├── OS
                                 │   ├── Python ABI
                                 │   └── SHA256
                                 │
                                 └── excluded
                                 ├── .env
                                 ├── tests/
                                 ├── docs/internal/
                                 └── sensitive .py sources
```

Commandes utiles

```
# Build wheel
                        python -m pip install build
                        python -m build

                        # Inspect wheel content
                        python -m zipfile -l dist/*.whl

                        # Install into clean venv
                        python -m venv /tmp/package_test
                        source /tmp/package_test/bin/activate
                        python -m pip install dist/*.whl

                        # Verify compiled module
                        python -c "import package.protected_core.scoring; print('compiled import OK')"
```

Python, PyPI, versions et dependances

Les problemes Nuitka sont souvent lies a l'environnement Python : mauvaise version, mauvais venv, ABI differente, dependances installees globalement, ou package natif incompatible. Ces liens aident a stabiliser la base Python.

Ressource	URL	Usage	Commande utile
PyPI Nuitka	https://pypi.org/project/Nuitka/	Verifier installation et versions publiees.	`python -m pip install -U nuitka`
Python Downloads	https://www.python.org/downloads/	Installer une version Python cible.	`python --version`
pip documentation	https://pip.pypa.io/	Installation packages, contraintes, index.	`python -m pip --version`
venv documentation	https://docs.python.org/3/library/venv.html	Environnements virtuels propres.	`python -m venv .venv`

Commandes diagnostic Python

```
python --version
                                python -c "import sys; print(sys.version)"
                                python -c "import sys; print(sys.executable)"
                                python -m pip --version
                                python -m pip list
                                python -m nuitka --version
```

A surveiller

Point	Pourquoi
Version Python	Les modules `.pyd` / `.so` dependent de l'ABI.
Architecture	Windows amd64, Linux x86_64, ARM64 ne sont pas interchangeables.
venv actif	Evite de compiler avec les mauvaises dependances.
Dependances natives	DLL/SO/DYLIB peuvent casser au runtime.

GitHub Nuitka : code source, issues, discussions

GitHub est utile pour chercher des bugs connus, lire des issues, verifier les pull requests, consulter les discussions techniques et comprendre les comportements recents. Avant d'ouvrir une issue, il faut toujours chercher le message d'erreur exact.

Ressource	URL	Usage
Repository Nuitka	https://github.com/Nuitka/Nuitka	Code source, README, branches, releases.
Issues	https://github.com/Nuitka/Nuitka/issues	Rechercher bugs, erreurs connues, cas similaires.
Releases GitHub	https://github.com/Nuitka/Nuitka/releases	Versions publiees cote GitHub.
Pull requests	https://github.com/Nuitka/Nuitka/pulls	Changements en cours, corrections futures.

Template d'information pour issue

```
# Minimal diagnostic information

                        ## Nuitka version
                        python -m nuitka --version

                        ## Python version
                        python -c "import sys; print(sys.version); print(sys.executable)"

                        ## OS / architecture
                        python -c "import platform; print(platform.platform()); print(platform.machine())"

                        ## Command used
                        python -m nuitka ...

                        ## Error
                        Paste complete build log or runtime traceback.

                        ## Minimal reproduction
                        Attach a tiny example if possible.
```

> **Regle de courtoisie technique :** une issue utile contient toujours un exemple minimal, la commande exacte, la version Nuitka et le log complet.

CI/CD, GitHub Actions et releases

Un build Nuitka devient industriel quand il est reproductible hors machine developpeur. CI/CD permet de tester plusieurs OS, plusieurs versions Python, de generer des artefacts, de stocker les logs et d'attacher les releases a un tag Git.

Ressource	URL	Usage
GitHub Actions documentation	https://docs.github.com/actions	Configurer workflows CI/CD.
actions/setup-python	https://github.com/actions/setup-python	Installer Python dans un workflow GitHub Actions.
actions/upload-artifact	https://github.com/actions/upload-artifact	Publier les artefacts de build.
GitHub Releases	https://docs.github.com/repositories/releasing-projects-on-github	Attacher builds, zips, wheels et notes de version.

Workflow minimal Linux

```
name: nuitka-build-linux

                        on:
                        workflow_dispatch:
                        push:
                        tags:
                        - "v*"

                        jobs:
                        build:
                        runs-on: ubuntu-latest

                        steps:
                        - uses: actions/checkout@v4

                        - name: Install system packages
                        run: |
                        sudo apt update
                        sudo apt install -y build-essential ccache patchelf zip

                        - uses: actions/setup-python@v5
                        with:
                        python-version: "3.12"

                        - name: Install dependencies
                        run: |
                        python -m pip install --upgrade pip setuptools wheel
                        python -m pip install -r requirements-build.txt

                        - name: Run tests
                        run: |
                        python -m pytest -v

                        - name: Build Nuitka
                        run: |
                        mkdir -p build/logs build/reports
                        python -m nuitka \
                        --mode=standalone \
                        --output-dir=build/nuitka \
                        --report=build/reports/report.xml \
                        --assume-yes-for-downloads \
                        app.py 2>&1 | tee build/logs/build.log

                        - name: Upload artifact
                        uses: actions/upload-artifact@v4
                        with:
                        name: app-linux-standalone
                        path: |
                        build/nuitka/**
                        build/logs/**
                        build/reports/**
```

Protection du code, securite, limites

Nuitka peut proteger raisonnablement un coeur metier en evitant de livrer directement les sources Python sensibles. Mais ce n'est pas une garantie d'inviolabilite. Les secrets, cles, tokens et licences maitres ne doivent jamais etre codes en dur.

Ressources utiles

Ressource	URL	Usage
Nuitka User Manual	https://nuitka.net/user-documentation/user-manual.html	Comprendre ce que Nuitka compile et comment distribuer.

Ressource	URL	Usage
Python secrets management	https://12factor.net/config	Ne pas coder les secrets en dur, utiliser configuration externe.
OWASP Secrets Management	https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html	Bonnes pratiques de gestion des secrets.
Code signing Windows	https://learn.microsoft.com/windows/win32/seccrypto/cryptography-tools	Signature de binaires Windows pour distribution professionnelle.

A retenir

```
Protection raisonnable
                                │
                                ├── Compiler protected_core
                                ├── Ne pas livrer les .py sensibles
                                ├── Garder une API publique minimale
                                ├── Inspecter la wheel finale
                                ├── Ne jamais coder de secrets en dur
                                ├── Signer les binaires si distribution Windows
                                └── Deplacer les fonctions ultra-critiques cote serveur si besoin
```

Attention : un secret dans un binaire reste un secret expose. Nuitka protege du copier-coller facile, pas des mauvaises pratiques de securite.

Parcours de lecture recommande

Cette derniere section donne un parcours de lecture en fonction de l'objectif. Elle permet a un utilisateur de ne pas se perdre dans toutes les pages et de lire uniquement ce qui correspond a son probleme immediat.

Objectif utilisateur	Lire en priorite	Ensuite	Resultat attendu
Installer Nuitka	User Manual	Tutorial setup and build	Venv propre, Nuitka installe, version verifiee.
Premier build	Tutorial setup and build	User Manual command line usage	Script minimal compile et execute.
Standalone	User Manual	Common issue solutions	Dossier `.dist` fonctionnel hors repo.
Onefile	User Manual	Common issues, antivirus Windows	Executable unique teste apres standalone.
Compiler un addon Django	User Manual module mode	Python packaging guide	Module `.pyd` / `.so` importable.
Data files / templates	User Manual data files	Setuptools package data	Templates, static, JSON, YAML inclus correctement.
Probleme runtime	Common issue solutions	GitHub Issues	Erreur classee et correctif cible.
Release professionnelle	GitHub Actions docs	Python packaging guide	CI/CD, artefacts, checksums, release notes.

Parcours debutant

```
1. Site officiel
                                2. User Manual
                                3. Tutorial setup and build
                                4. Premier hello.py
                                5. Standalone
                                6. Common issues si probleme
```

Parcours professionnel

```
1. User Manual
                                2. Package configuration
                                3. Common issue solutions
                                4. Build script versionne
                                5. Tests source/binaire
                                6. CI/CD
                                7. Release inspectee
```

Conclusion : Nuitka se maitrise progressivement. Les meilleures ressources sont celles qui aident a repondre a une question precise : installer, compiler, inclure des fichiers, diagnostiquer, packager, puis industrialiser.