Projet d'examen — Système multi-langages (C · Python · Flask · C#)

Projet d'examen intégrateur en 5 composantes : une DLL en C, un module Python testé avec Pytest, une API REST Flask avec dashboard Chart.js, et une interface C# — à réaliser individuellement sur l'un des 6 sujets au choix.

5TTR 6TTR Exercice ambitieux

Tu vas réaliser un projet complet qui fait collaborer **quatre langages** autour d'un même domaine métier : une bibliothèque C compilée en DLL, un module Python qui l'exploite, une API Flask qui sert les données, et une interface C# qui les affiche — le tout complété d'un dashboard web et d'un générateur de données.

🎯 Objectifs

À la fin de ce projet, tu seras capable de :

1. Compiler un fichier C en bibliothèque partagée (`.dll`) et l'appeler depuis Python via `ctypes`.
2. Écrire un module Python testable avec **Pytest** (minimum 10 tests PASSED).
3. Concevoir une API REST avec **Flask** exposant des données au format JSON.
4. Créer un **dashboard web** interactif avec Chart.js consommant l'API.
5. Développer une interface **C# Windows Forms** communicant avec l'API via `HttpClient`.
6. Mesurer et comparer les performances de trois approches de calcul (Python pur, ctypes scalaire, ctypes batch).

Partie 1 — Architecture générale

La chaîne technique relie 5 composantes. Chaque couche ne communique qu'avec celle directement en dessous ou au-dessus : **le C n'est jamais appelé directement par Flask**.

© 2025 Mr. Lambrechts


```
calculs.c  →  gcc -shared -o calculs.dll calculs.c -lm
                        ↓  ctypes
              module.py  ←──  pytest tests_module.py
                        ↓
              app.py  (Flask)
                ├── /api/...        ← JSON ←── GUI C# (HttpClient)
                └── /dashboard/   ← HTML + Chart.js (lecture seule)

              generator.py  ──POST──►  /api/...
```

💡 **RÈGLE D'OR :** tout calcul reste dans la DLL ou le module Python. La GUI C# et le dashboard *affichent* — ils ne calculent rien.

Compilation de la DLL

```bash
# Linux / macOS
gcc -shared -fPIC -o calculs.so calculs.c -lm
```

© 2025 Mr. Lambrechts

```
# Windows (MinGW)
gcc -shared -o calculs.dll calculs.c -lm

# Vérifier les fonctions exportées (Linux)
nm -D calculs.so | grep " T "
```

Chargement depuis Python

```python
import ctypes

lib = ctypes.CDLL("./calculs.dll")

# Déclarer les types — OBLIGATOIRE pour éviter les erreurs silencieuses
lib.calc_bmi.restype  = ctypes.c_float
lib.calc_bmi.argtypes = [ctypes.c_float, ctypes.c_float]

# Appel scalaire
bmi = lib.calc_bmi(ctypes.c_float(70.0), ctypes.c_float(1.75))

# Appel batch (tableau de N valeurs)
N = 1_000_000
weights = (ctypes.c_float * N)(*liste_poids)
heights = (ctypes.c_float * N)(*liste_tailles)
results = (ctypes.c_float * N)()
lib.calc_bmi_batch(weights, heights, results, N)
```

Partie 2 — Les 6 projets

Choisis **un seul** projet. Télécharge le cahier des charges complet via les liens ci-dessus.

#	Projet	Domaine	Fonctions C (exemples)
1	Station de mesures	Physique / capteurs	`to_fahrenheit`, `is_anomaly`
2	Suivi fitness	Santé / sport	`calc_bmi`, `calories_burned`
3	Tournoi ELO	Jeux / classement	`expected_score`, `new_elo`
6	Caisse POS	Commerce	`calc_vat`, `render_change`

© 2025 Mr. Lambrechts

#	Projet	Domaine	Fonctions C (exemples)
8	Notes scolaires	Éducation	`weighted_avg`, `std_deviation`
10	Calculatrice scientifique	Mathématiques	`calc_gcd`, `calc_factorial`

> 💡 **POUR LES 5E :** persistance en JSON (dossier `data/`). **Pour les 6e :** base de données SQL (MySQL via Laragon ou SQLite).

Partie 3 — Les 5 composantes en détail

3.1 — Module C (`calculs.c` → `calculs.dll`)

Deux niveaux de fonctions pour chaque projet :

Type	Utilisation	Exemple
Scalaire	Application normale	`float calc_bmi(float w, float h)`
Batch	Benchmark uniquement	`void calc_bmi_batch(float* w, float* h, float* out, int n)`

3.2 — Module Python (`module.py`)

Wrapper `ctypes` + fonctions métier testables. Flask l'importe directement — pas de logique dans `app.py`.

```python
# module.py — exemple pour le projet 1
import ctypes, os

_lib = ctypes.CDLL(os.path.join(os.path.dirname(__file__), "calculs.dll"))
_lib.to_fahrenheit.restype  = ctypes.c_float
_lib.to_fahrenheit.argtypes = [ctypes.c_float]

def to_fahrenheit(celsius: float) -> float:
    return _lib.to_fahrenheit(ctypes.c_float(celsius))

def validate_measure(value: float, sensor_type: str) -> bool:
    ranges = {"temperature": (-50, 100), "pressure": (800, 1100), "humidity
```

© 2025 Mr. Lambrechts

```
        lo, hi = ranges.get(sensor_type, (float('-inf'), float('inf')))
        return lo <= value <= hi
```

3.3 — API Flask (`app.py`)

Toutes les réponses respectent le format uniforme :

```python
from flask import Flask, jsonify, request
import module  # le module Python local

app = Flask(__name__)

@app.route("/api/measures", methods=["GET"])
def get_measures():
    data = module.load_all()  # lit JSON ou SQL
    return jsonify({"status": "ok", "data": data})

@app.route("/api/measures", methods=["POST"])
def add_measure():
    body = request.json
    if not module.validate_measure(body["value"], body["type"]):
        return jsonify({"status": "error", "message": "Valeur hors plage"})
    result = module.save_measure(body)
    return jsonify({"status": "ok", "data": result}), 201
```

3.4 — Dashboard Web (`/dashboard`)

Accessible sur `http://localhost:5000/dashboard`. Lecture seule, 3 graphiques Chart.js, rafraîchissement automatique.

```
<!-- templates/dashboard.html -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
  async function loadData() {
    const res  = await fetch('/api/measures/stats');
    const json = await res.json();
    // Mise à jour des graphiques avec json.data
  }
  loadData();
  setInterval(loadData, 30_000); // rafraîchit toutes les 30 secondes
</script>
```

© 2025 Mr. Lambrechts

3.5 — Interface C# (`MainForm.cs`)

Toute la communication passe par `HttpClient`. Aucun calcul dans le code C#.

```
using System.Net.Http;
using System.Text;
using System.Text.Json;

private readonly HttpClient _client = new()
    { BaseAddress = new Uri("http://localhost:5000") };

// GET
private async Task<JsonDocument> GetAsync(string path) {
    var resp = await _client.GetAsync(path);
    var json = await resp.Content.ReadAsStringAsync();
    return JsonDocument.Parse(json);
}

// POST
private async Task PostAsync(string path, object body) {
    var json    = JsonSerializer.Serialize(body);
    var content = new StringContent(json, Encoding.UTF8, "application/json"
    await _client.PostAsync(path, content);
}
```

Partie 4 — Générateur de données automatique

Le générateur simule des utilisateurs réels en envoyant des données à ton API. Il utilise **Python +** `requests`.

Active d'abord CORS dans Flask (nécessaire si le générateur tourne dans un autre processus) :

```
pip install flask-cors
```

```
# app.py — ajouter
from flask_cors import CORS
CORS(app)
```

Script `generator.py`

```
"""
    Générateur automatique de données — Projets TTINFO 5e/6e
    Usage : python generator.py --projet 1 --n 100 --delay 0.3
"""
import requests, random, time, argparse
from datetime import date, timedelta

BASE = "http://localhost:5000"

# ── Générateurs par projet ──────────────────────────────────────

def gen_projet1(n, delay):
    """Station de mesures physiques"""
    types   = ['temperature', 'pressure', 'humidity']
    ranges  = {'temperature':(-10,45), 'pressure':(950,1060), 'humidity':(1
    units   = {'temperature':'°C', 'pressure':'hPa', 'humidity':'%'}
    for i in range(n):
        t   = random.choice(types)
        lo, hi = ranges[t]
        val = round(random.uniform(lo, hi * (1.15 if random.random()<.08 el
        r   = requests.post(f"{BASE}/api/measures", json={"type":t,"value":
        _log(i+1, n, r.status_code, f"{t}={val}")
        time.sleep(delay)

def gen_projet2(n, delay):
    """Suivi sportif & fitness"""
    types = ['course', 'vélo', 'natation', 'musculation', 'yoga']
    mets  = {'course':8.0, 'vélo':6.0, 'natation':7.0, 'musculation':4.0, '
    for i in range(n):
        t  = random.choice(types)
        dur = random.randint(20, 90)
        r  = requests.post(f"{BASE}/api/sessions", json={
            "type": t, "duration_min": dur,
            "met_value": mets[t] + random.uniform(-0.5, 0.5),
            "date": str(date.today() - timedelta(days=random.randint(0,30))
        })
        _log(i+1, n, r.status_code, f"{t} {dur}min")
        time.sleep(delay)

def gen_projet3(n, delay):
    """Tournoi ELO — génère des matchs entre joueurs existants"""
```

```
    players = requests.get(f"{BASE}/api/players").json().get("data", [])
    if len(players) < 2:
        print("Ajoute d'abord au moins 2 joueurs via la GUI.")
        return
    for i in range(n):
        a, b = random.sample(players, 2)
        res  = random.choice([1.0, 0.5, 0.0])
        r    = requests.post(f"{BASE}/api/matches",
               json={"player_a_id":a["id"],"player_b_id":b["id"],"result":r
        _log(i+1, n, r.status_code, f"{a['name']} vs {b['name']} → {res}")
        time.sleep(delay)

def gen_projet6(n, delay):
    """Caisse POS — génère des transactions"""
    products = requests.get(f"{BASE}/api/products").json().get("data", [])
    if not products:
        print("Ajoute d'abord des produits via la GUI.")
        return
    for i in range(n):
        items = [{"product_id": p["id"], "qty": random.randint(1,4)}
                 for p in random.sample(products, k=min(random.randint(1,4)
        r = requests.post(f"{BASE}/api/transactions",
            json={"items": items, "amount_given": 50.0})
        _log(i+1, n, r.status_code, f"{len(items)} article(s)")
        time.sleep(delay)

def gen_projet8(n, delay):
    """Notes scolaires"""
    students = requests.get(f"{BASE}/api/students").json().get("data", [])
    courses  = requests.get(f"{BASE}/api/courses").json().get("data", [])
    if not students or not courses:
        print("Ajoute d'abord des élèves et des cours.")
        return
    for i in range(n):
        s = random.choice(students)
        c = random.choice(courses)
        r = requests.post(f"{BASE}/api/grades", json={
            "student_id": s["id"], "course_id": c["id"],
            "value": round(random.gauss(12, 3.5), 1),
            "weight": random.choice([1.0, 1.0, 1.0, 2.0]),
            "date": str(date.today())
        })
        _log(i+1, n, r.status_code, f"{s['name']} — {c['name']}")
        time.sleep(delay)
```

```python

def gen_projet10(n, delay):
    """Calculatrice scientifique"""
    ops = ["sqrt({})", "power({},{})", "gcd({},{})", "factorial({})", "lcm(
    for i in range(n):
        expr = random.choice(ops)
        if expr.count('{}') == 1:
            expr = expr.format(random.randint(1, 144))
        else:
            a, b = random.randint(1,50), random.randint(1,50)
            expr = expr.format(a, b)
        r = requests.post(f"{BASE}/api/calculate", json={"expression": expr
        _log(i+1, n, r.status_code, expr)
        time.sleep(delay)

# ── Utilitaire ──────────────────────────────────────────────────────────

def _log(i, n, code, info):
    ok = "✅" if code in (200,201) else "❌"
    print(f"  [{i:>4}/{n}] {ok} HTTP {code} | {info}")

GENS = {1:gen_projet1, 2:gen_projet2, 3:gen_projet3,
        6:gen_projet6, 8:gen_projet8, 10:gen_projet10}

# ── CLI ─────────────────────────────────────────────────────────────────

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Générateur de données TTI
    parser.add_argument("--projet", type=int, choices=[1,2,3,6,8,10], requi
                        help="Numéro du projet (1/2/3/6/8/10)")
    parser.add_argument("--n",      type=int,   default=50,
                        help="Nombre d'entrées à générer (défaut: 50)")
    parser.add_argument("--delay",  type=float, default=0.2,
                        help="Délai entre chaque requête en secondes (défau
    parser.add_argument("--url",    default="http://localhost:5000",
                        help="URL de base de l'API Flask")
    args = parser.parse_args()

    BASE = args.url
    print(f"\n🚀 Générateur — Projet {args.projet} — {args.n} entrées @ {a
    print(f"   API cible : {BASE}\n")
    GENS[args.projet](args.n, args.delay)
    print(f"\n✅ Terminé.")
```

© 2025 Mr. Lambrechts

Utilisation

```
# Installer la dépendance
pip install requests

# Générer 100 mesures pour le projet 1, une toutes les 0.3 secondes
python generator.py --projet 1 --n 100 --delay 0.3

# Générer 50 matchs pour le projet 3 (joueurs déjà créés)
python generator.py --projet 3 --n 50 --delay 0.1

# Pointer vers une API distante
python generator.py --projet 6 --n 200 --url http://192.168.1.10:5000
```

> 💡 **ASTUCE :** lance d'abord ton API Flask (`python app.py`), puis le générateur dans un second terminal. Observe le dashboard se remplir en temps réel.

Partie 5 — Benchmark de performance

Le fichier `benchmark.py` mesure trois approches sur **10 batches de 1 000 000 calculs**.

Scénario	Description	Résultat attendu
A — Python pur	Boucle Python native	Référence (×1.0)
B — ctypes scalaire	1 appel ctypes par itération	**Plus lent** que A (overhead ×N)
C — ctypes batch	1 seul appel, C traite tout	Le plus rapide (×10 à ×20)

```
# benchmark.py — squelette à compléter
import ctypes, time

lib = ctypes.CDLL("./calculs.dll")
# TODO : déclarer argtypes et restype

N = 1_000_000
NB_BATCHES = 10

```

```python
def bench(label, fn):
    times = []
    for _ in range(NB_BATCHES):
        t = time.perf_counter()
        fn()
        times.append(time.perf_counter() - t)
    total = sum(times)
    return label, total, total / NB_BATCHES

# TODO : implémenter A(), B(), C()
# TODO : afficher le tableau avec les facteurs ×

# Format de sortie attendu :
# Scénario              Temps total    Moy./batch    Facteur
# Python pur            0.08 s         8.0 ms        ×1.0
# ctypes scalaire       0.83 s        83.0 ms        ×0.1   ⚠️
# ctypes batch          0.006 s        0.6 ms        ×13    ✅
```

Question orale attendue : *"Explique pourquoi le scénario B est plus lent que Python pur."* Réponse attendue : l'overhead `ctypes` (conversion de types, franchissement du GIL) est plus coûteux que le calcul lui-même. La solution : déplacer la **boucle dans le C**, pas uniquement le calcul.

🧪 Consignes de remise

Structure de dossier attendue

```
projet-XXXX/
├── calculs.c              ← code source C
├── calculs.dll            ← DLL compilée
├── module.py              ← wrapper ctypes + fonctions métier
├── tests_module.py        ← 10 tests Pytest (tous PASSED)
├── app.py                 ← API Flask + route /dashboard
├── generator.py           ← générateur de données
├── benchmark.py           ← benchmark 3 scénarios
├── templates/
│   └── dashboard.html     ← dashboard Chart.js
├── data/                  ← (5e seulement) fichiers JSON
│   └── *.json
```

© 2025 Mr. Lambrechts

```
└── GUI/
    └── ProjetXX.sln        ← solution C# Windows Forms
```

Barème synthétique

Composante	Points
Module C (scalaire + batch, compile sans erreur)	/4
Module Python (wrapper + fonctions)	/4
Tests Pytest (10/10 PASSED)	/3
API Flask (tous les endpoints)	/4
Dashboard web (3 graphiques Chart.js)	/3
GUI C# Windows Forms	/4
Benchmark (3 scénarios + tableau + explication orale)	/2
Total	**/24**

Exercice 1 — Vérifie ta chaîne technique

Avant de tout développer, vérifie que la chaîne de base fonctionne :

1. Écris `calculs.c` avec **une seule fonction** (ex. `float add(float a, float b)`)
2. Compile-la en DLL
3. Appelle-la depuis Python avec `ctypes` et affiche le résultat
4. Lance Flask et crée un endpoint `GET /test` qui appelle ce module Python
5. Depuis C# : appelle `GET /test` et affiche la réponse dans un Label

Une fois cette mini-chaîne validée, tu peux ajouter les vraies fonctions.

Exercice 2 — Lance le générateur et observe le dashboard

1. Lance ton API Flask
2. Lance le générateur : `python generator.py --projet XX --n 50 --delay 0.5`
3. Ouvre le dashboard dans le navigateur
4. Observe les graphiques se remplir en temps réel
5. Note ce qui se passe si l'API est arrêtée pendant la génération

---

✍️ À retenir

© 2025 Mr. Lambrechts

- Le **C** est compilé en DLL et appelé via `ctypes` — jamais directement depuis Flask.
- La **boucle** doit être dans le C (batch) pour de vraies performances — les appels `ctypes` répétés N fois sont plus lents que Python pur.
- Le **module Python** est le seul pont entre la DLL et le reste de l'application.
- Le **dashboard** est lecture seule — il affiche, il ne modifie jamais de données.
- La **GUI C#** ne recalcule rien — elle délègue tout à l'API via `HttpClient`.
- Les **tests Pytest** testent le module Python, ce qui teste indirectement le C.

Suite

Le projet est à remettre complet avec tous les fichiers listés ci-dessus. Prépare une **démonstration de 5 minutes** : lance l'API, le générateur, montre le dashboard en direct, puis la GUI C#. Explique ta chaîne technique et les résultats du benchmark.

© 2025 Mr. Lambrechts