Projet de groupe
Un plugin Claude Code pour les situations d'urgence

Christophe Guyeux

IUT NFC – UMLP

12 mai 2026

Sommaire

Le projet et pourquoi des skills

L'objectif du projet

Vous allez concevoir, en groupe, un plugin Claude Code capable d'interroger toutes les informations utiles dans le cadre de situations d'urgence :

- Localiser un site, des routes, des accès à l'eau, des bâtiments.
- Caractériser une zone (population, vulnérabilités, équipements sensibles).
- Connaître l'état d'alerte sanitaire, épidémiologique, météorologique.

Livrable

Un plugin Claude Code installable, regroupant un ensemble cohérent de skills qu'un agent peut activer à la demande pour aider à la décision en situation tendue.

Pourquoi des skills (et pas des MCPs) ?

La question d'architecture se pose tout de suite : ces capacités, faut-il les exposer comme MCP servers ou comme skills ?

Les deux savent rendre une capacité disponible à Claude, mais leur coût en contexte est radicalement différent.

L'enjeu

En situation d'urgence, on veut un agent réactif, qui ne consomme pas la moitié de son contexte à se présenter lui-même avant qu'on ne lui ait rien demandé.

Le coût caché des MCPs

Quand un MCP est déclaré, Claude voit en permanence :

- Le nom du serveur.
- Pour chaque outil : nom, signature, docstring complète.
- Plus le nombre d'outils est grand, plus c'est lourd.

Pour un MCP géo qui expose 5 outils avec docstrings détaillées : $\approx 1\,500$ tokens.

Avec une trentaine de sources potentielles « utiles en urgence », on frôle ou dépasse 50 000 tokens en pure infrastructure, avant la moindre question.

Comment les skills changent la donne

Avec un skill, Claude ne voit, par défaut :

- Que le nom et la description (50 tokens par skill).
- Le contenu de `SKILL.md` se charge *seulement* si la description matche la requête.
- Les ressources annexes (scripts, références) ne se chargent qu'à la demande, depuis `SKILL.md`.

Trois ordres de grandeur

Un MCP : 1 000-2 000 tokens consommés en permanence.
Un skill inactif : 50 tokens.
Un skill actif : 500-2 000 tokens, mais *seulement quand utilisé*.

Les autres avantages

- Pas de serveur à démarrer : zéro processus à superviser, zéro port à libérer.
- Pas de hub central à maintenir.
- Plus simple à débugger : un script Python testable seul.
- Distribution : un dossier zip ou un repo Git, installable par `cp` ou via plugin.
- Versioning : chaque skill a son propre cycle de vie, indépendamment des autres.

Le revers de la médaille

- Le format n'est pas standardisé entre éditeurs : ChatGPT, Gemini et Claude Code ont chacun leur propre mécanisme de skill, avec des conventions et des emplacements différents. Un MCP, lui, suit un standard ouvert consommable par tout client compatible (Cursor, Cline, etc.).
- Pas de transport HTTP natif : un skill local s'exécute sur la machine du client, pas en service.
- Pour des cas multi-clients ou API exposée, le MCP reste pertinent.

Choix retenu pour le projet

On vise une assistance locale, économe en contexte, déployable sur le terminal d'un opérateur : skills + plugin Claude Code, pas MCP.

Méthodologie : créer un skill

Construire un skill, étape par étape

Pour chaque capacité à exposer à Claude :

1. Cadrer : que doit-on pouvoir demander en langage naturel ?
2. Identifier la donnée et la voie d'accès (API, fichier, requête).
3. Écrire la logique dans un script CLI Python testable seul.
4. Rédiger le `SKILL.md` : frontmatter + body.
5. Soigner la `description` : c'est elle qui déclenche le routage.
6. Restreindre les `allowed-tools` au strict nécessaire.
7. Tester l'auto-déclenchement avec des questions naturelles.
8. Mesurer l'empreinte tokens (idle / actif).

Cadrer une capacité utile en urgence

Avant d'écrire la moindre ligne de code, pour chaque skill candidat :

- Quel besoin opérationnel en situation d'urgence (3-4 questions-types).
- Quelle source de données (API publique, fichier, requête web).
- Quelle sortie attendue, dans quel format synthétique.
- Quelles dépendances (bibliothèques Python, binaires externes).

Critère

Si un skill ne sert qu'en exploration de fond, il n'a rien à faire dans ce plugin. Seules les capacités mobilisables en urgence ont leur place.

Écrire la logique du skill

Un skill peut prendre plusieurs formes selon la nature de la capacité :

- Pur Markdown : un `SKILL.md` seul, qui décrit une procédure, des règles, un *prompt* de cadrage. Aucun code à exécuter.
- Markdown + script CLI : un `SKILL.md` qui appelle un `main.py` (ou un binaire), pour interroger une API, traiter un fichier, etc.
- Markdown + références : un `SKILL.md` court qui pointe vers des fichiers `references/` chargés à la demande.

Pour ce projet

La plupart des skills interrogeront des sources externes (APIs, fichiers), donc viendront avec un script `main.py` en Python ordinaire, testable unitairement. Pas de framework, pas de serveur, pas de port à exposer.

Bien rédiger la `description`

C'est le seul levier pour que le skill soit auto-déclenché.

Bonnes pratiques :

- Énoncer les phrases-types : « Trigger when user asks ... ».
- Citer les mots-clés métier : noms de datasets, terminologie du domaine.
- Inclure les variantes : français et anglais, abréviations (BPE, IRIS, INSEE).
- Pas plus de 200 mots, l'essentiel en premier.

Anti-pattern

« Skill pour interroger des données » : Claude ne le déclenchera jamais.

Restreindre les `allowed-tools`

Par défaut, un skill activé hérite des outils de la session. On peut le restreindre :

- `allowed-tools: Bash(python3 *)` : seulement l'exécution Python.
- `allowed-tools: Read, Grep` : skill de lecture pure.
- `allowed-tools: Bash(curl -get https://api.exemple.fr/*)` : pattern réseau précis.

Sécurité

Restreindre = défense en profondeur : si un skill est mal écrit, il ne peut pas faire des dégâts au-delà de ses outils whitelistés.

Tester l'auto-déclenchement

1. Installer le skill : `cp -r mon-skill ~/.claude/skills/`.
2. Démarrer une session `claude` dans un dossier neutre.
3. Poser une question naturelle qui devrait matcher la description.
4. Vérifier dans la trace que Claude charge bien le skill.
5. Si non, ajuster la description, recommencer.
6. Tester aussi le déclenchement explicite : `/<nom-skill>`.

MCP versus skill

Cas d'étude : un petit MCP `time`

Pour visualiser la différence MCP / skill sur quelque chose de petit et lisible, prenons un MCP `time` :

- Deux outils : `get_current_time`, `convert_time`.
- Pas de base de données, pas d'API externe.
- Logique pure : juste des conversions `ZoneInfo`.

Voyons côté à côté ce que donne un même besoin exprimé en MCP ou en skill. (Ce n'est pas un MCP qui fera partie de votre périmètre, juste une illustration.)

Version MCP : time_mcp.py

```python
# mymcps/time_mcp.py -- version actuelle (MCP server)

from datetime import datetime, timedelta
from zoneinfo import ZoneInfo
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("time")


@mcp.tool()
def get_current_time(timezone: str | None = None) -> dict:
    """Renvoie l'heure courante dans le fuseau IANA demandé."""
    tz = ZoneInfo(timezone or "Europe/Paris")
    now = datetime.now(tz)
    jan = now.replace(month=1, day=15)
    is_dst = bool((now.utcoffset() or timedelta()) != (jan.utcoffset() or timedelta()))
    return {"timezone": tz.key, "datetime": now.isoformat(), "is_dst": is_dst}


@mcp.tool()
def convert_time(source_timezone: str, time: str, target_timezone: str) -> dict:
    """Convertit une heure entre deux fuseaux."""
    hh, mm = map(int, time.strip().split(":", 1))
    src_tz = ZoneInfo(source_timezone)
    tgt_tz = ZoneInfo(target_timezone)
    today = datetime.now(src_tz).date()
    src_dt = datetime(today.year, today.month, today.day, hh, mm, tzinfo=src_tz)
    tgt_dt = src_dt.astimezone(tgt_tz)
    return {"source": src_dt.isoformat(), "target": tgt_dt.isoformat()}


if __name__ == "__main__":
    mcp.run(transport="streamable-http")

# Empreinte tokens : ~30 lignes effectives + boilerplate FastMCP.
# Charge dans le contexte LLM dès que le serveur est listé, même si non utilisé.
```

Version skill : SKILL.md

```
---
name: time
description: Renvoie l'heure courante d'un fuseau IANA, ou convertit une heure entre deux fuseaux.
    Trigger when user asks "quelle heure est-il à...", "quelle heure il sera à...", "heure locale", "
    convertir HH:MM Paris → Tokyo", ou mentionne un fuseau IANA.
allowed-tools: Bash(python3 *)
---

# Skill `time`

Pour l'heure courante :
```bash
python3 ${CLAUDE_SKILL_DIR}/time.py current --timezone "$ARGUMENTS"
```

Pour une conversion :
```bash
python3 ${CLAUDE_SKILL_DIR}/time.py convert --from "Europe/Paris" --time "14:00" --to "Asia/Tokyo"
```

La sortie est du JSON ; reformate-la en phrase naturelle avant de répondre.
Pour les options avancées (DST, format custom), voir `references/options.md`.
```

Version skill : `time.py` (logique pure)

```python
#!/usr/bin/env python3
"""Script du skill time. Pas de FastMCP, pas de serveur. Logique pure."""
import argparse, json, sys
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo


def is_dst(dt):
    jan = dt.replace(month=1, day=15)
    return bool((dt.utcoffset() or timedelta()) != (jan.utcoffset() or timedelta()))


def cmd_current(tz_name):
    tz = ZoneInfo(tz_name or "Europe/Paris")
    now = datetime.now(tz)
    return {"timezone": tz.key, "datetime": now.isoformat(), "is_dst": is_dst(now)}


def cmd_convert(src, time, tgt):
    hh, mm = map(int, time.strip().split(":", 1))
    src_tz, tgt_tz = ZoneInfo(src), ZoneInfo(tgt)
    today = datetime.now(src_tz).date()
    src_dt = datetime(today.year, today.month, today.day, hh, mm, tzinfo=src_tz)
    tgt_dt = src_dt.astimezone(tgt_tz)
    return {"source": src_dt.isoformat(), "target": tgt_dt.isoformat()}


if __name__ == "__main__":
    p = argparse.ArgumentParser()
    sub = p.add_subparsers(dest="cmd", required=True)
    pc = sub.add_parser("current"); pc.add_argument("--timezone")
    pv = sub.add_parser("convert")
    pv.add_argument("--from", dest="src", required=True)
    pv.add_argument("--time", required=True)
    pv.add_argument("--to", dest="tgt", required=True)
    a = p.parse_args()
    out = cmd_current(a.timezone) if a.cmd == "current" else cmd_convert(a.src, a.time, a.tgt)
    print(json.dumps(out, ensure_ascii=False))
```

Comparaison chiffrée

```
## Comparaison MCP `time_mcp.py` vs skill `time/`

### Avant (MCP)
- Serveur HTTP/streamable-http qui tourne en permanence (port 8001+).
- Outils `get_current_time` et `convert_time` enregistrés via `@mcp.tool()`.
- Listés dans `.mcp.json` du projet : Claude voit leur signature **dès le démarrage de session**, même s'il ne les
    utilise jamais.
- Empreinte tokens : ~250 tokens listés en permanence (nom + signature + docstring).

### Après (skill)
- Pas de serveur, juste un script CLI invoqué à la demande.
- `SKILL.md` (~25 lignes) référencé via sa `description`.
- Empreinte tokens **inactif** : ~50 tokens (juste nom + description).
- Empreinte tokens **actif** (chargé) : 250 tokens à l'invocation, 0 le reste du temps.

### Gain typique sur 30 MCPs
Si on a 30 MCPs en mémoire permanente : ~7,500 tokens consommés en pure attente.
Avec 30 skills en progressive disclosure : ~1,500 tokens en attente, le reste à la demande.
**Économie : ~6,000 tokens par session.** À l'échelle d'une journée, c'est plusieurs euros et de la latence en moins.
```

Domaines à couvrir

Le périmètre du plugin

Le plugin doit donner accès, en situation d'urgence, à tout ce qui aide à comprendre un territoire et décider vite. Quelques grandes familles :

- Géographie et accessibilité.
- Population et vulnérabilités.
- Équipements sensibles et établissements à risque.
- Veille sanitaire et épidémiologique.
- Visualisation et restitution.
- Briques utilitaires transverses (temps, météo, séries).

Exemples de domaines

La liste ci-dessous est indicative. Vous pouvez :

- Compléter avec d'autres capacités utiles (énergie, transports, télécoms...).
- Remplacer un domaine par un autre si votre groupe a une meilleure idée.
- Réagencer le découpage : ce ne sont que des suggestions de regroupement.

1. Géographie : BD Topo, BD Forêt, IRIS, OSM, accès eau, bâti.
2. Démographie : population, familles, activité résidents, état civil.
3. Socio-économique : revenus, logement, suroccupation, navettes, emploi.
4. Équipements : BPE, ICPE/SEVESO, hébergement, scolaire.
5. Santé / épidémio : Sentinelles, attente CH, recherche PDF.
6. Visualisation : Mermaid, tableaux interactifs.
7. Utilitaires : temps, séries temporelles, météo, vigilances.

Sources de données : risques, santé, météo

Domaine	Sources / URLs
BPE (équipements)	https://www.insee.fr/fr/statistiques/3568638
ICPE / SEVESO	https://www.georisques.gouv.fr/
Sentinelles (épidémio)	https://www.sentiweb.fr/
Santé publique France	https://geodes.santepubliquefrance.fr/
Eau (Hub'Eau)	https://hubeau.eaufrance.fr/
Vigicrues	https://www.vigicrues.gouv.fr/
Qualité de l'air (Géod'air)	https://www.geodair.fr/
Météo-France (API)	https://portail-api.meteofrance.fr/
OpenMeteo (libre)	https://open-meteo.com/
BisonFuté (trafic)	https://www.bison-fute.gouv.fr/

Sources de données : géo et social

Domaine	Sources / URLs
Géographie (IGN)	https://data.geopf.fr/ (Géoplateforme WFS/WMS)
OSM / Overpass	https://overpass-api.de/
Panoramax	https://panoramax.fr/
Adresses (BAN)	https://api-adresse.data.gouv.fr/
Géo communes/IRIS	https://geo.api.gouv.fr/
INSEE (population, familles, logement, revenus, navettes, emploi, formation)	https://api.insee.fr/
Atlas statistique INSEE	https://statistiques-locales.insee.fr/

Choisir et compléter

Chaque groupe a une liberté de cadrage dans son domaine :

- Choisir les capacités les plus utiles en situation d'urgence.
- Découper en skills (un skill par capacité, ou un skill multi-fonctions selon le besoin de routage).
- Compléter si une capacité importante manque (ex : skill météo court terme).
- Remplacer par d'autres sources jugées plus pertinentes (autres APIs, autres données ouvertes).
- Justifier le découpage dans le `SKILL.md` ou un `references/architecture.md`.

Mot d'ordre

Pas de quota : 5 skills bien faits valent mieux que 12 vagues.

Réduire l'empreinte tokens

Pourquoi c'est l'enjeu central

En production, le contexte est cher et fini :

- Coût direct : tokens facturés à chaque appel API.
- Coût en latence : plus de tokens = plus de temps avant la première réponse.
- Coût en qualité : un contexte rempli de descriptifs noie l'attention du modèle.

Objectif chiffré

À nombre de capacités égales, diviser par 3 ou plus l'empreinte tokens *idle* (sans question posée) par rapport à l'approche « tout en MCPs ».

Trois leviers principaux

1. Soigner la `description` : la plus courte possible, mais explicite et bien ciblée. C'est elle qui sera *toujours* chargée.
2. Sortir le détail dans `references/` : tout ce qui est doc avancée ne doit charger qu'à la demande.
3. Externaliser les gros contenus : datasets, exemples longs, configs YAML lourdes ne doivent pas être inline dans `SKILL.md`.

Levier 1 : description courte mais utile

- Ne pas répéter le nom du skill : il est déjà connu via `name`.
- Pas de « This skill will help you ... » : aller directement aux phrases-types qui déclenchent.
- Garder une liste de mots-clés métier, pas trois phrases redondantes.

Avant / après

Avant (180 tokens) : trois paragraphes mal structurés.
Après (60 tokens) : une phrase « Trigger when ... » + 6 mots-clés. Mêmes performances de routage.

Levier 2 : `references/` pour le détail

Structure type d'un skill complexe :

- `SKILL.md` (court) : quand l'utiliser, comment lancer le script, où trouver le détail.
- `references/api.md` : doc des endpoints, paramètres acceptés, codes d'erreur.
- `references/exemples.md` : cas d'usage avec entrées/sorties.
- `references/troubleshooting.md` : erreurs courantes, comment les diagnostiquer.

Tant que Claude n'a pas explicitement *besoin* de la doc API, elle reste sur disque. Coût zéro.

Levier 3 : externaliser les gros contenus

- Les fichiers de configuration longs (200+ lignes) ne doivent jamais être inline dans `SKILL.md`.
- Les datasets de référence (codes IRIS, codes COG) restent en CSV / Parquet, lus par le script.
- Les schémas JSON Schema (entrées d'outils) peuvent vivre en `references/schema.json` et être inclus seulement si Claude doit les valider.

Livrables et évaluation

Modalités de rendu

- Projet à réaliser en groupe (3 à 4 étudiants).
- Rendu sous forme d'un dépôt GitHub public.
- Le lien du dépôt (`https://github.com/<orga>/<repo>`) doit être envoyé par mail à `guyeux@gmail.com`.
- Échéance : vendredi 12 juin 2026, fin de journée.

Important

Un seul mail par groupe, avec en objet `[Projet Skills] Groupe N - nom1, nom2, ...` et le lien GitHub en corps de message. Tout retard ou dépôt privé non accessible au correcteur entraîne une pénalité.

Références et ressources

Documentation Claude Code

- Quickstart : `https://code.claude.com/docs/fr/quickstart`.
- Documentation skills : `https://code.claude.com/docs/fr/skills`.
- Skill bundling dans plugins : `https://code.claude.com/docs/fr/plugins`.
- Cours Claude Code : 9 chapitres dont le 4 entièrement sur les skills.

Outils utiles

- `/skill-creator` : skill natif qui aide à créer, tester et optimiser un skill.
- `/context` : affiche les tokens consommés en temps réel. Indispensable pour mesurer.
- `/agents` : liste les sous-agents disponibles.
- `/help` : voir tous les skills disponibles dans la session.

Bibliothèques Python utiles selon le domaine

- Géo : `requests`, `owslib`, `shapely`, `pyproj`.
- INSEE : `pynsee`, `requests` sur l'API INSEE.
- PDF / extraction : `pypdf`, `pdfplumber`, `rank-bm25`.
- Mermaid : binaire `mmdc` (Mermaid CLI).
- Météo : APIs Météo-France, OpenMeteo.

À déclarer dans le `requirements.txt` ou `pyproject.toml` de votre plugin.

Connaissances de fond

- INSEE : codes IRIS, codes COG, principaux datasets (RP, Filosofi, BPE, Flores).
- Géoportail : services WFS / WMS, produits IGN (BD Topo, BD Forêt).
- Métier : terminologie spécifique au domaine, structure des données opérationnelles.
- OpenStreetMap : Overpass QL, taggage des cours d'eau et routes.

Ne pas réinventer

La plupart de ces concepts sont déjà documentés en ligne (INSEE.fr, IGN, Géoportail, OSM Wiki). Lisez avant de coder.

Bonne chance !

Questions par mail ou en TP.