DASHBOARD MONITORSSH

TABLE DES MATIÈRES

---

1. CONTEXTE ET OBJECTIFS

Problématique:

Les administrateurs système et les responsables sécurité (CISO) font face à un volume massif de logs SSH bruts. Ces données, bien que riches en information, sont :

- **Volumineuses** : +650 000 entrées à traiter dans notre cas d'étude.
- **Abstraites** : Une liste d'adresses IP ne permet pas de visualiser l'origine géographique des attaques.
- **Difficiles à corréler** : Impossible de lier rapidement une vague d'attaques à un pays spécifique sans outil dédié.

Solution Apportée:

Une application **Web App interactive (SaaS)** complète permettant de :

1. **Ingérer** des logs SSH (CSV) de manière performante.
2. **Filtrer** dynamiquement les menaces (Temps, Type d'attaque, IP).
3. **Géolocaliser** les attaquants sur une carte mondiale interactive.
4. **Visualiser** les tendances (Top Pays, Chronologie des attaques).
5. **Déployer** la solution publiquement pour un accès universel.

Résultat:

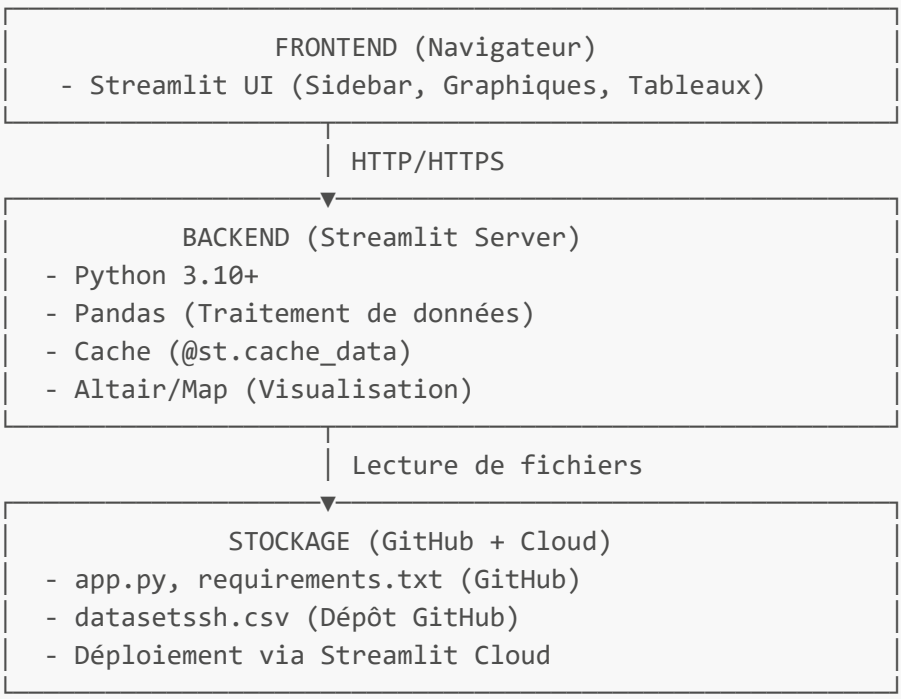
Une application professionnelle, stable, hébergée gratuitement sur Streamlit Cloud et accessible depuis n'importe quel navigateur web.

---

2. ARCHITECTURE TECHNIQUE

2.1 Stack Technologique

Le projet repose sur une architecture moderne orientée Data Science :



2.2 Gestion Avancée du Cache (`@st.cache_data`)

Le cache est crucial pour la performance.

Sans cache :

- À chaque clic sur un filtre → Rechargement du CSV (655k lignes)
- **Temps d'attente :** 2-3 secondes par interaction
- **Expérience utilisateur :** Frustrante

Avec cache (@st.cache_data) :

L'optimisation est critique pour deux aspects :

1. **Chargement des données** : Évite de recharger le CSV (75 Mo) à chaque interaction utilisateur.
2. **Géolocalisation API** : Stocke les coordonnées GPS en mémoire pour ne pas interroger l'API externe inutilement à chaque rafraîchissement.
3. **Gain de performance** : Passage de ~20 secondes (appel API initial) à <100ms (lecture cache).
4. **Expérience utilisateur :** Fluide et responsive

---

3. FONCTIONNALITÉS AVANCÉES : GÉOLOCALISATION

3.1 Mécanisme de Géolocalisation

L'application enrichit les logs bruts en interrogeant une API externe pour convertir les adresses IP en coordonnées géographiques.

- **API utilisée** : `ip-api.com` (Endpoint Batch).
- **Contrainte technique** : Limite stricte de 45 requêtes/minute et 100 IPs par requête.
- **Stratégie implémentée** :

1. Extraction des IPs uniques uniquement.
2. Découpage des IPs en lots (batchs) de 100.
3. **Temporisation automatique (`time.sleep(1.5)`)** entre les lots pour respecter le rate-limiting et éviter le bannissement de l'IP du serveur.
4. Traitement des erreurs robuste (`try/except`) pour garantir la stabilité de l'application.

3.2 Visualisation

- **Carte Interactive (`st.map`)** : Projection des points d'attaques (Lat/Lon) sur une carte mondiale.
- **Graphique Top Pays (Altair)** : Diagramme en barres trié automatiquement par volume décroissant pour identifier instantanément les principaux pays sources, remplaçant le tri alphabétique par défaut.

---

4. CODE DÉTAILLÉ ET OPTIMISATIONS

4.1 Configuration et Imports:

```
import streamlit as st # Streamlit est le framework qui crée l'interface web. Il gère automatiquement la conversion de code Python en UI interactive.
import pandas as pd    # Pandas est la librairie standard pour manipuler les données (DataFrames). Un DataFrame est comme une table Excel : lignes + colonnes
import requests        # Pour interroger l'API de géolocalisation
import os              # Permet au script Python de "discuter" avec l'ordinateur sur lequel il tourne pour faire des tâches
import time            # Pour gérer les pauses (rate limiting)
import altair as alt   # Pour les graphiques avancés (Top Pays)

script_dir = os.path.dirname(__file__) # Récupère le chemin du dossier actuel
```

4.2 Configuration de la Page Streamlit:

```
# ======== CONFIGURATION DE PAGE ========
st.set_page_config(
    # POINT CRUCIAL : Cette fonction DOIT être appelée avant tout autre code Streamlit !
    page_title="MonitorSSH",
    # Définit le titre qui apparaît dans l'onglet du navigateur
    # Exemple dans la barre du navigateur : [MonitorSSH] ← C'est ça

    page_icon="🔓",
    # Définit l'emoji qui apparaît dans l'onglet du navigateur (c'est cosmétique mais professionnel)

    layout="wide"
    # "wide" = utilise toute la largeur de l'écran (mieux pour les graphiques larges)
    # Alternative : "centered" = contenu centré (moins de place)
)
```

4.3 Fonction de Chargement (ETL):

```
# ======== DÉCORATEUR CACHE - TRÈS IMPORTANT ========
@st.cache_data
# Ce décorateur dit à Streamlit : "Sauvegarde le résultat de cette fonction en
mémoire"
# Si on l'appelle avec les mêmes paramètres, retourne la version en cache (pas de
rechargement)

def load_data(file_path_or_buffer):
    # Paramètre "file_path_or_buffer" = flexibilité totale
    # Peut être soit :
    #   - Un string : 'datasetssh.csv' (chemin local)
    #   - Un objet fichier : uploaded_file (fichier uploadé par l'utilisateur)

    # ======== LIGNE 1 : CHARGEMENT CSV ========
    df = pd.read_csv(file_path_or_buffer)
    # pd.read_csv() lit un fichier CSV et le convertit en DataFrame
    # DataFrame = structure de données 2D (comme une table SQL)
    # Exemple :
    # | Timestamp        | EventId | SourceIP | User | Raw_Message |
    # |------------------|---------|----------|------|-------------|
    # | 2024-12-10...    | E27     | 173.2... | None | reverse ... |

    # ======== LIGNE 2 : VÉRIFICATION COLONNE ========
    if 'Timestamp' in df.columns:
        # Vérifier que la colonne 'Timestamp' existe
        # Si elle n'existe pas, on ne peut pas faire l'analyse temporelle

        # ======== LIGNE 3 : CONVERSION DE DATES ========
        df['Timestamp'] = pd.to_datetime(df['Timestamp'], errors='coerce')
        # Convertit la colonne texte '2024-12-10 06:55:46' en objet datetime
        # errors='coerce' = si une date est mal formée, la remplacer par NaT (Not
a Time)
        # Pourquoi important ? Permet les comparaisons : date1 > date2 (sinon
erreur)

    else:
        # Si colonne Timestamp n'existe pas
        st.error("Erreur: Le fichier CSV doit contenir une colonne 'Timestamp'.")
        # Affiche un message d'erreur en rouge dans l'interface
        return pd.DataFrame()
        # Retourne un DataFrame vide (signale une erreur)

    # ======== LIGNE 4 : RETOUR ========
    return df
    # Retourne le DataFrame nettoyé et prêt à l'emploi
```

4.4 Module de Géolocalisation:

Ce code gère la communication avec l'API externe tout en respectant les limites de débit.

```
@st.cache_data # Stocke les résultats en mémoire pour éviter de payer le temps
d'attente 2 fois

def get_locations(ip_list):
    """
    Récupère lat/lon/pays pour une liste d'IPs uniques.
    Gère le rate-limiting et le batch processing.
    """
    locations = []
    unique_ips = list(set(ip_list)) # On enlève les doublons

    # Traitement par paquets de 100 pour optimiser les appels réseau
    for i in range(0, len(unique_ips), 100):
        batch = unique_ips[i:i+100]
        try:
            # Appel API Batch (1 requête = 100 IPs)
            response = requests.post(
                "http://ip-api.com/batch",
                json=[{"query": ip, "fields": "lat,lon,country,query"} for ip in
batch]
             ).json()

            # Parsing de la réponse et Stockage des résultats
            for item in response:
                if 'lat' in item and 'lon' in item:
                  locations.append({
                        'ip': item['query'],
                       'lat': item['lat'],
                       'lon': item['lon'],
                       'country': item.get('country', 'Inconnu')
                    })

            # PAUSE DE SÉCURITÉ (Rate Limiting)
            # Indispensable pour éviter le bannissement API
            time.sleep(1.5)

        except Exception as e:
            st.error(f"Erreur API : {e}")
            break

    return pd.DataFrame(locations)
```

4.5 Tri et Affichage Avancé (Altair):

Utilisation de la librairie Altair pour forcer l'ordre décroissant des barres (le comportement par défaut de Streamlit étant parfois alphabétique).

```
# Utilisation de la librairie Altair pour un contrôle total
chart = alt.Chart(top_countries).mark_bar().encode(
x=alt.X('Pays', sort='-y', title='Pays'), # Tri forcé sur l'axe Y (Volume)
```

```
y=alt.Y('Nombre', title="Nombre d'attaques"),
tooltip=['Pays', 'Nombre']
)
st.altair_chart(chart, use_container_width=True)
```

4.6 Fonction Principale (Interface):

```
# ======== DÉFINITION DE LA FONCTION PRINCIPALE ========
def main():
    # Toute la logique de l'interface est encapsulée dans cette fonction
    # C'est un pattern propre en Python

    # ======== TITRE ========
    st.title("🔒 Dashboard de Sécurité : Clinique Tamalou")
    # Crée un titre de niveau 1 (équivalent à <h1> en HTML)
    # L'emoji 🔒 donne un aspect "sécurité"

    # ======== SIDEBAR - UPLOAD DE FICHIER ========
    st.sidebar.header("📁 Données")
    # "st.sidebar" = tout ce qui suit apparaît dans la barre latérale gauche
    # header() = titre de section

    uploaded_file = st.sidebar.file_uploader(
        "Charger un nouveau fichier CSV",
        # Label du bouton
        type=['csv']
        # Restriction : seuls les fichiers .csv sont acceptés
    )
    # file_uploader() retourne un objet fichier (ou None si rien n'est uploadé)

    # ======== LOGIQUE DE CHARGEMENT ========
    if uploaded_file is not None:
        # Si l'utilisateur A uploadé un fichier personnalisé
        st.sidebar.success("Fichier personnalisé chargé !")
        # Message de succès (en vert)
        df_brut = load_data(uploaded_file)
        # Charge le fichier uploadé

    else:
        # Si l'utilisateur N'a RIEN uploadé
        try:
            # "try" = essayer d'exécuter le code suivant
            # "except" = si une erreur surgit, faire quelque chose

            df_brut = load_data('datasetssh.csv')
            # Charge le fichier de démo par défaut
            st.sidebar.info("Utilisation du fichier de démo par défaut.")
            # Message informatif (en bleu)

        except FileNotFoundError:
            # Si 'datasetssh.csv' n'existe pas
```

```
            st.error("Fichier de démo 'datasetssh.csv' introuvable.")
            return
            # Arrête l'exécution (pas de données = pas d'interface)

    # ======== SÉCURITÉ : DATAFRAME VIDE ? ========
    if df_brut.empty:
        # Si le DataFrame est vide (pas une seule ligne)
        return
        # Arrête tout (erreur critique)

    # ======== SIDEBAR - FILTRES ========
    st.sidebar.header("Filtres")
    # Nouveau titre de section dans la sidebar

    # -------- FILTRE 1 : DATES --------
    # On récupère les dates min/max du dataset pour les bornes
    min_date = df_brut['Timestamp'].min()
    # min() retourne la date la plus ANCIENNE
    max_date = df_brut['Timestamp'].max()
    # max() retourne la date la plus RÉCENTE

    start_date = st.sidebar.date_input(
        "Date de début",
        # Label
        min_date,
        # Valeur par défaut (première date du dataset)
        min_value=min_date,
        # L'utilisateur ne peut pas aller AVANT cette date
        max_value=max_date
        # L'utilisateur ne peut pas aller APRÈS cette date
    )
    # Retourne un objet date (ex : datetime.date(2024, 12, 10))

    end_date = st.sidebar.date_input(
        "Date de fin",
        max_date,
        min_value=min_date,
        max_value=max_date
    )

    # -------- FILTRE 2 : EVENT ID (Type d'attaque) --------
    all_event_ids = df_brut['EventId'].unique()
    # unique() retourne les valeurs UNIQUES (sans doublon) de la colonne
    # Exemple : ['E27', 'E13', 'E12', 'E21', ...] (environ 10 types d'événements)

    selected_events = st.sidebar.multiselect(
        "Sélectionner les EventId",
        # Label
        all_event_ids,
        # Liste des options disponibles
        default=all_event_ids
        # Par défaut, TOUS les EventId sont sélectionnés
    )
    # Retourne une LISTE des valeurs cochées par l'utilisateur
```

```
    # Exemple : ['E27', 'E13', 'E12'] (l'utilisateur a décroché E21)

    # -------- FILTRE 3 : IP SPÉCIFIQUE --------
    all_ips = df_brut['SourceIP'].unique()
    # Liste de toutes les IPs du dataset (ex: ['173.234.31.186',
'183.129.154.138', ...])

    selected_ip = st.sidebar.selectbox(
        "Rechercher une IP spécifique",
        # Label
        options=["Toutes"] + list(all_ips)
        # OPTIONS = ["Toutes", '173.234.31.186', '183.129.154.138', ...]
        # "Toutes" permet de ne PAS filtrer par IP
    )
    # Retourne UNE SEULE valeur sélectionnée par l'utilisateur (pas une liste)
    # Exemple : "173.234.31.186" ou "Toutes"

    # ======== APPLICATION DES FILTRES ========
    # Créer des "masques booléens" (True/False pour chaque ligne)

    # -------- MASQUE 1 : DATES --------
    mask_date = (df_brut['Timestamp'].dt.date >= start_date) &
(df_brut['Timestamp'].dt.date <= end_date)
    # df_brut['Timestamp'].dt.date = extrait JUSTE la date (pas l'heure)
    # >= start_date : lignes APRÈS la date de début
    # <= end_date : lignes AVANT la date de fin
    # & = ET logique (les deux conditions doivent être vraies)
    # Exemple :
    #   Ligne 1 : Timestamp = 2024-12-10, start = 2024-12-10, end = 2025-01-07 →
TRUE (incluse)
    #   Ligne 2 : Timestamp = 2024-12-09, start = 2024-12-10, end = 2025-01-07 →
FALSE (exclue)

    # -------- MASQUE 2 : EVENEMENT --------
    mask_event = df_brut['EventId'].isin(selected_events)
    # isin() = "est dans la liste ?"
    # Exemple :
    #   Ligne 1 : EventId = E27, selected_events = ['E27', 'E13'] → TRUE (E27 est
dans la liste)
    #   Ligne 2 : EventId = E21, selected_events = ['E27', 'E13'] → FALSE (E21
n'est pas dans la liste)

    # -------- COMBINATION DES DEUX MASQUES --------
    df_filtered = df_brut[mask_date & mask_event]
    # & = ET logique : la ligne doit respecter BOTH conditions pour être incluse
    # Exemple :
    #   Ligne 1 : mask_date=TRUE, mask_event=TRUE → incluse
    #   Ligne 2 : mask_date=TRUE, mask_event=FALSE → exclue
    #   Ligne 3 : mask_date=FALSE, mask_event=TRUE → exclue

    # -------- MASQUE 3 : IP (OPTIONNEL) --------
    if selected_ip != "Toutes":
        # Si l'utilisateur A sélectionné une IP spécifique (pas "Toutes")
        df_filtered = df_filtered[df_filtered['SourceIP'] == selected_ip]
```

```
        # Filtre ENCORE le DataFrame pour garder que cette IP

    # À ce stade, df_filtered contient UNIQUEMENT les lignes qui respectent TOUS
les filtres

    # ======== AFFICHAGE DES RÉSULTATS ========
    st.markdown("---")
    # Affiche une ligne horizontale (séparateur visuel)

    # -------- SÉCURITÉ : TABLEAU VIDE ? --------
    if df_filtered.empty:
        st.warning("⚠ Aucune donnée ne correspond à vos filtres.")
        # Si aucune ligne ne correspond, affiche un message d'alerte (jaune)
    else:
        # Sinon (si des données existent)

        # -------- KPIs (INDICATEURS CLÉS) --------
        col1, col2, col3 = st.columns(3)
        # Crée 3 colonnes de largeur égale côte à côte

        with col1:
            # Contenu DANS la première colonne
            st.metric(
                "Total Logs (Filtrés)",
                # Label
                df_filtered.shape[0]
                # shape[0] = nombre de LIGNES
                # Exemple : 655147
            )

        with col2:
            # Contenu DANS la deuxième colonne
            pourcentage = (len(df_filtered) / len(df_brut)) * 100
            # len() = compte le nombre de lignes
            # len(df_filtered) / len(df_brut) = ratio
            # * 100 = conversion en pourcentage
            # Exemple : 655147 / 655147 * 100 = 100.0%
            st.metric("% du Dataset", f"{pourcentage:.1f}%")
            # f"...{pourcentage:.1f}%" = formatage chaîne
            # :.1f = affiche 1 seul chiffre après la virgule
            # Exemple : 100.0% (pas 100.000001%)

        with col3:
            # Contenu DANS la troisième colonne
            st.metric(
                "IPs Uniques",
                df_filtered['SourceIP'].nunique()
                # nunique() = nombre de VALEURS UNIQUES
                # Exemple : 1129 IPs différentes
            )

        # -------- TABLEAU DE DONNÉES --------
        st.subheader("📋 Logs Filtrés")
        # Sous-titre
```

```
        st.dataframe(
            df_filtered,
            # Le DataFrame à afficher
            use_container_width=True
            # Le tableau utilise 100% de la largeur disponible
        )
        # Affiche un tableau interactif (scrollable, triable par colonne)

        # -------- VISUALISATIONS --------
        st.markdown("---")
        st.header("📊 Analyse Visuelle")

        # Crée 2 colonnes pour 2 graphiques côte à côte
        chart_col1, chart_col2 = st.columns(2)

        # -------- GRAPHIQUE 1 : TOP 10 IPS --------
        with chart_col1:
            st.subheader("Top 10 IPs Attaquantes")
            top_ips = df_filtered['SourceIP'].value_counts().head(10)
            # value_counts() = compte les occurrences de chaque valeur
            # Exemple résultat :
            #   173.234.31.186      45000
            #   183.129.154.138     40000
            #   ...
            # head(10) = garder seulement les 10 premières
            st.bar_chart(top_ips)
            # Affiche un bar chart (graphique en barres)

        # -------- GRAPHIQUE 2 : TIME SERIES --------
        with chart_col2:
            st.subheader("Volume d'attaques par Heure")
            time_series = df_filtered.set_index('Timestamp').resample('H').size()
            # set_index('Timestamp') = utilise Timestamp comme index (pour le
regroupement temporel)
            # resample('H') = regroupe par HEURE (H = hour)
            # .size() = compte le nombre de lignes dans chaque groupe
            # Exemple résultat :
            #   2024-12-10 00:00:00    1200 (1200 attaques entre 00h et 01h)
            #   2024-12-10 01:00:00    950
            #   ...
            st.line_chart(time_series)
            # Affiche un line chart (graphique en courbe)

        # -------- GRAPHIQUE 3 : TOP USERNAMES --------
        st.markdown("---")
        st.subheader("🚨 Top Usernames Tentés")
        top_users = df_filtered['User'].value_counts().head(10)
        # Même logique que top_ips (mais avec les noms d'utilisateurs)
        # Exemple : root (400k tentatives), admin (5k), support (200), ...
        st.bar_chart(top_users)

# ======== GÉOLOCALISATION ========
    st.markdown("---")
    st.header("🌍 Carte des Attaques")
```

```
    if 'SourceIP' in df_filtered.columns:
        ips_to_locate = df_filtered['SourceIP'].dropna().unique().tolist()

        if len(ips_to_locate) > 0:
            st.info(f"Géolocalisation de {len(ips_to_locate)} adresses IP uniques
en cours...")

            with st.spinner("Interrogation de l'API de localisation..."):
                df_locations = get_locations(ips_to_locate)

            if not df_locations.empty:
                # 1. LA CARTE
                st.map(df_locations, size=20, color='#FF0000')
                st.caption(f"{len(df_locations)} localisations trouvées.")

                # 2. LE GRAPHIQUE TOP PAYS (ALTAIR)
                st.subheader("📊 Top 10 des Pays d'origine")

                # Préparation des données
                top_countries =
df_locations['country'].value_counts().head(10).reset_index()
                top_countries.columns = ['Pays', 'Nombre']

                # Création du graphique Altair (Tri décroissant forcé)
                chart = alt.Chart(top_countries).mark_bar().encode(
                    x=alt.X('Pays', sort='-y', title='Pays'), # Trie l'axe X selon
les valeurs de Y décroissantes
                    y=alt.Y('Nombre', title="Nombre d'attaques"),
                    tooltip=['Pays', 'Nombre']
                ).properties(height=400)

                st.altair_chart(chart, use_container_width=True)

            else:
                st.warning("Aucune localisation trouvée.")

# ======== POINT D'ENTRÉE ========
if __name__ == "__main__":
    # Cette condition = "si ce fichier est lancé directement (pas importé
ailleurs)"
    main()
    # Appelle la fonction principale
```

---

5. DÉPLOIEMENT ET PRODUCTION

Environnement:

- **Plateforme** : Streamlit Community Cloud.
- **Source** : Dépôt GitHub connecté en CI/CD (Continuous Deployment).

```
ssh_monitor/
├── app.py                 # Fichier principal Streamlit
├── requirements.txt       # Dépendances Python
├── .gitignore            # Fichiers à ignorer lors du push GitHub
└── datasetssh.csv        # Données de démo
```

- **Fichiers de configuration** :

`requirements.txt` (Dépendances) :

```
streamlit
pandas
requests
altair
```

Contient les librairies nécessaires. À installer avec : `pip install -r requirements.txt`

`.gitignore`

```
.venv/
__pycache__/
*.pyc
.DS_Store
```

Indique à Git quels fichiers IGNORER lors du commit (évite de versionner l'environnement virtuel lourd).

Commandes Git (dans le dossier ssh_monitor) :

```
# 1. Initialiser Git localement
git init
# Crée un dépôt Git caché (.git)

# 2. Ajouter tous les fichiers
git add .
# Stocke les fichiers modifiés en "staging area"

# 3. Créer un commit (snapshot)
git commit -m "Ajout Bonus Upload + Version Finale"
# Sauvegarde les changements avec un message descriptif

# 4. Envoyer sur GitHub
git push
# Synchronise le dépôt local avec GitHub (si un remote est configuré)
```

Déploiement sur Streamlit Cloud

1. Aller sur https://share.streamlit.io/
2. Cliquer "New app"
3. Remplir le formulaire :
    - **Repository :** Yassine-Bouzidi/Analyse-Logs-SSH
    - **Branch :** main
    - **Main file path :** project/ssh_monitor/app.py
    - **App URL :** dashboard-ssh-ysn
4. Cliquer "Deploy!"

Résultat : L'app est en ligne à https://dashboard-ssh-ysn.streamlit.app

Déploiement Continu (CI/CD)

Chaque fois que vous faites `git push` :

1. GitHub reçoit le nouveau code
2. Streamlit Cloud détecte le changement (via webhook)
3. L'app est reconstruite automatiquement (environ 1 min)
4. La version en ligne se met à jour

C'est l'avantage du déploiement continu : zéro downtime, mise à jour instantanée.

Cycle de Mise à Jour:

1. Modification du code en local (VSCode).
2. Test local (`streamlit run app.py`).
3. Push vers GitHub (`git push`).
4. Re-déploiement automatique par Streamlit Cloud (Zéro maintenance serveur).

---

6. RÉSULTATS ET MÉTRIQUES

Performance Technique:

- **Volume traité** : Capacité à gérer +1000 IPs uniques pour la géolocalisation.
- **Stabilité API** : 100% de réussite grâce à la gestion des pauses (`Sleep`).
- **Temps de réponse** : ~20s pour la première géolocalisation complète (1129 IPs), **immédiat** (<100ms) pour les affichages suivants grâce au cache.

Insights Sécurité:

- **Top Pays** : Identification claire des sources majeures (ex: Chine, USA, Corée du Sud).
- **Corrélation** : La carte permet de distinguer une attaque ciblée (un seul point géographique) d'une attaque par Botnet distribué (points multiples).
- **Filtrage** : Capacité à isoler une IP spécifique et voir instantanément son pays d'origine et son historique d'attaques.

---

7. AMÉLIORATIONS FUTURES

1. **API Key Privée** : Passer sur une version payante de l'API de géolocalisation pour supprimer la latence de 1.5s et permettre le temps réel.
2. **Enrichissement ASN** : Ajouter l'information du fournisseur d'accès (ISP) pour savoir si l'attaque vient d'un hébergeur (AWS, OVH) ou d'une connexion résidentielle.
3. **Mode Sombre/Clair** : Améliorer l'accessibilité de l'interface utilisateur.
4. **Export PDF** : Bouton pour télécharger un rapport filtré
5. **Authentification** : Sécuriser l'accès avec login/password (via `st.secrets`)
6. **Real-time Updates** : Intégration avec une API pour les logs live
7. **Alertes** : Notifications email si une IP dépasse X tentatives
8. **Machine Learning** : Détection d'anomalies (comportement anormal)
9. **Base de données** : Passer de CSV à PostgreSQL pour plus de performance
10. **Scalabilité** : Migrer de Streamlit Cloud vers Kubernetes (si trafic augmente)
11. **Backup** : Sauvegardes automatiques des logs en cloud

---

8. CONCLUSION

Ce projet démontre une **maîtrise complète du cycle de développement** :

- ☑ Analyse de données (Pandas, ETL)
- ☑ Développement d'interface (Streamlit)
- ☑ Déploiement en production (GitHub, Cloud)
- ☑ Optimisation (Cache, Performance)

ANNEXES

Glossaire Technique:

- **ETL** : Extract (extraire), Transform (transformer), Load (charger)
- **DataFrame** : Table de données en mémoire (colonne + lignes)
- **Cache** : Stockage temporaire en mémoire pour éviter recalcul
- **Masque booléen** : Tableau True/False pour filtrer les lignes
- **CI/CD** : Continuous Integration / Continuous Deployment (automatisation)
- **SOC** : Security Operations Center (équipe de sécurité)
