🔧 Guide Pratique : Intégrer les IA Open Source dans PaxNet

Exemples Concrets par Système pour la Double Mission

📌 Vue d'Ensemble des Intégrations

mermaid

```
graph TD
    A[ETHOS] -->|Valide| B[NEXUS]
    B -->|Orchestre| C[CHRONOS]
    B -->|Coordonne| D[HARMONY]
    C -->|Alerte| E[PHOENIX]
    D -->|Mobilise| F[ATLAS]

    subgraph "IA Open Source"
        G[TensorFlow]
        H[Transformers]
        I[Scikit-learn]
        J[Prophet]
        K[OpenCV]
    end

    C -.->|Utilise| G
    C -.->|Utilise| J
    D -.->|Utilise| H
    F -.->|Utilise| I
    E -.->|Utilise| K
```

🔴 ETHOS : Intégration SHAP + Fairlearn

Installation

bash

```
pip install shap fairlearn lime tensorflow interpretability
```

Code d'Intégration Complet

python

```python
# ethos/ai_integration/explainable_validator.py
import shap
from fairlearn.metrics import MetricFrame, selection_rate
from fairlearn.reductions import ExponentiatedGradient, DemographicParity
import numpy as np
from typing import Dict, Any, List, Tuple

class ExplainableEthicalValidator:
    """
    Validateur éthique avec IA explicable pour double mission
    """

    def __init__(self):
        self.model = self._load_ethics_model()
        self.explainer = shap.Explainer(self.model)
        self.fairness_constraint = DemographicParity()

    def validate_with_explanation(self,
                                  action: Dict[str, Any],
                                  context: Dict[str, Any]) -> Tuple[bool, Dict]:
        """
        Valide une action avec explication complète
        Pour PAIX et CATASTROPHES
        """
        # 1. Préparer features double mission
        features = self._extract_features(action, context)

        # 2. Prédiction avec le modèle
        prediction = self.model.predict([features])[0]

        # 3. Explication SHAP
        shap_values = self.explainer(np.array([features]))

        # 4. Vérification équité
        fairness_check = self._check_fairness(action, context)

        # 5. Décision finale avec transparence
        approved = (
            prediction > 0.8 and  # Haute confiance
            fairness_check['is_fair'] and
            self._respects_dual_mission(action)
        )

        return approved, {
            'prediction_score': float(prediction),
            'explanation': self._format_shap_explanation(shap_values),
```

```python
            'fairness_metrics': fairness_check,
            'dual_mission_check': {
                'serves_peace': self._serves_peace_mission(action),
                'serves_disaster': self._serves_disaster_mission(action)
            },
            'top_factors': self._get_top_decision_factors(shap_values),
            'confidence': self._calculate_confidence(prediction, fairness_check)
        }

    def _extract_features(self, action: Dict, context: Dict) -> np.ndarray:
        """Extraction features pour double mission"""
        return np.array([
            # Features communes
            action.get('urgency_level', 0),
            action.get('affected_people', 0),
            action.get('has_consent', 0),

            # Features mission paix
            context.get('conflict_intensity', 0),
            context.get('mediation_readiness', 0),

            # Features mission catastrophe
            context.get('disaster_severity', 0),
            context.get('evacuation_needed', 0),

            # Features éthiques
            action.get('transparency_score', 0),
            action.get('privacy_protection', 0)
        ])

    def _check_fairness(self, action: Dict, context: Dict) -> Dict:
        """Vérifie l'équité pour tous les groupes"""
        # Simuler vérification sur différents groupes
        groups = context.get('affected_groups', [])

        if not groups:
            return {'is_fair': True, 'details': 'No group data'}

        # Analyse équité
        outcomes = []
        for group in groups:
            outcome = self._simulate_outcome_for_group(action, group)
            outcomes.append(outcome)

        # Calculer disparité
        disparity = max(outcomes) - min(outcomes) if outcomes else 0
```

```python
        return {
            'is_fair': disparity < 0.1,  # Seuil 10%
            'disparity': disparity,
            'group_outcomes': dict(zip(groups, outcomes))
        }

# Exemple d'utilisation double mission
validator = ExplainableEthicalValidator()

# Cas 1 : Action de médiation
peace_action = {
    'type': 'mediation',
    'urgency_level': 0.6,
    'affected_people': 500,
    'has_consent': 1,
    'transparency_score': 0.9
}

# Cas 2 : Évacuation d'urgence
disaster_action = {
    'type': 'evacuation',
    'urgency_level': 0.95,
    'affected_people': 10000,
    'has_consent': 1,  # Consentement présumé en urgence
    'transparency_score': 1.0
}

# Validation avec explication
approved, explanation = validator.validate_with_explanation(
    peace_action,
    {'conflict_intensity': 0.7}
)
print(f"Médiation approuvée: {approved}")
print(f"Facteurs clés: {explanation['top_factors']}")
```

🟡 NEXUS : Intégration Hugging Face Transformers

Installation

```bash
pip install transformers torch langchain sentence-transformers
```

Orchestrateur Double Mission

python

```python
# nexus/ai_integration/dual_mission_orchestrator.py
from transformers import pipeline, AutoTokenizer, AutoModelForSeq2SeqLM
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
import torch

class DualMissionOrchestrator:
    """
    NEXUS utilise Transformers pour faciliter PAIX et CATASTROPHES
    """

    def __init__(self):
        # Modèles multilingues pour double mission
        self.translator = pipeline(
            "translation",
            model="facebook/nllb-200-distilled-600M"
        )

        self.sentiment_analyzer = pipeline(
            "sentiment-analysis",
            model="cardiffnlp/twitter-xlm-roberta-base-sentiment"
        )

        self.summarizer = pipeline(
            "summarization",
            model="facebook/bart-large-cnn"
        )

        # Modèle de génération pour propositions
        self.generator = self._init_generator()

    def generate_dual_proposals(self, crisis_data: Dict) -> List[Dict]:
        """
        Génère des propositions pour les DEUX missions
        """
        # 1. Analyser le contexte
        context_analysis = self._analyze_crisis_context(crisis_data)

        # 2. Générer propositions selon le type
        if crisis_data['type'] == 'conflict':
            proposals = self._generate_peace_proposals(crisis_data, context_analysis)
        elif crisis_data['type'] == 'disaster':
            proposals = self._generate_disaster_proposals(crisis_data, context_analysi
        else:  # Double crise
            proposals = self._generate_combined_proposals(crisis_data, context_analysi
```

```python
        # 3. Traduire pour toutes les langues affectées
        translated_proposals = self._translate_proposals(
            proposals,
            crisis_data.get('languages', ['en'])
        )

        # 4. Valider avec ETHOS
        validated = []
        for proposal in translated_proposals:
            if self.ethos_client.validate(proposal):
                validated.append({
                    **proposal,
                    'nexus_confidence': self._calculate_confidence(proposal),
                    'human_decision_required': True  # TOUJOURS
                })

        return validated

    def _generate_peace_proposals(self, crisis: Dict, analysis: Dict) -> List[Dict]:
        """Propositions spécifiques médiation"""
        prompt = f"""
        Context: {crisis['description']}
        Sentiment: {analysis['sentiment']}
        Cultural factors: {crisis.get('cultural_context', 'unknown')}

        Generate 3 mediation approaches that:
        1. Respect all parties
        2. Are culturally sensitive
        3. Focus on common ground
        4. Provide concrete next steps
        """

        approaches = self.generator(prompt, max_length=500, num_return_sequences=3)

        return [
            {
                'type': 'mediation_approach',
                'content': approach['generated_text'],
                'methodology': self._extract_methodology(approach),
                'estimated_duration': self._estimate_duration(approach),
                'required_resources': self._identify_resources(approach)
            }
            for approach in approaches
        ]

    def _generate_disaster_proposals(self, crisis: Dict, analysis: Dict) -> List[Dict]
        """Propositions spécifiques catastrophes"""
```

```python
        severity = crisis.get('severity', 'unknown')
        affected = crisis.get('affected_population', 0)

        proposals = []

        # Proposition 1 : Évacuation immédiate
        if severity == 'critical':
            proposals.append({
                'type': 'immediate_evacuation',
                'priority': 'maximum',
                'instructions': self._generate_evacuation_plan(crisis),
                'resources_needed': self._calculate_evacuation_resources(affected),
                'timeline': 'immediate'
            })

        # Proposition 2 : Abri sur place
        proposals.append({
            'type': 'shelter_in_place',
            'priority': 'high',
            'instructions': self._generate_shelter_instructions(crisis),
            'supplies_needed': self._calculate_shelter_supplies(affected),
            'duration_estimate': self._estimate_disaster_duration(crisis)
        })

        # Proposition 3 : Évacuation progressive
        if affected > 1000:
            proposals.append({
                'type': 'phased_evacuation',
                'priority': 'medium',
                'phases': self._plan_evacuation_phases(crisis, affected),
                'coordination': self._identify_coordination_needs(crisis)
            })

        return proposals

    def _translate_proposals(self, proposals: List[Dict], languages: List[str]) -> Lis
        """Traduit propositions dans toutes les langues nécessaires"""
        translated = []

        for proposal in proposals:
            translations = {'original': proposal}

            for lang in languages:
                if lang != 'en':  # Supposons que l'original est en anglais
                    translated_content = self.translator(
                        proposal.get('content', ''),
                        src_lang="eng_Latn",
```

```
                        tgt_lang=self._get_nllb_code(lang)
                    )

                    translations[lang] = {
                        **proposal,
                        'content': translated_content[0]['translation_text']
                    }

            translated.append(translations)

        return translated

# Exemple utilisation
orchestrator = DualMissionOrchestrator()

# Crise double : conflit + risque cyclone
double_crisis = {
    'type': 'combined',
    'description': 'Rising tensions in coastal area with approaching cyclone',
    'severity': 'high',
    'affected_population': 50000,
    'languages': ['en', 'fr', 'hi'],
    'conflict_parties': ['Group A', 'Group B'],
    'cyclone_category': 3,
    'estimated_landfall': '48 hours'
}

proposals = orchestrator.generate_dual_proposals(double_crisis)
print(f"Generated {len(proposals)} validated proposals for double crisis")
```

🟢 CHRONOS : Intégration TensorFlow + Prophet

Installation

```bash
pip install tensorflow prophet scikit-learn pandas numpy
```

Prédicteur Double Mission

python

```python
# chronos/ai_integration/dual_predictor.py
import tensorflow as tf
from prophet import Prophet
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from typing import Tuple, Dict

class DualMissionPredictor:
    """
    CHRONOS prédit conflits ET catastrophes avec ML/DL
    """

    def __init__(self):
        # Modèles pour conflits
        self.conflict_lstm = self._build_conflict_lstm()
        self.social_prophet = Prophet(
            changepoint_prior_scale=0.05,
            seasonality_mode='multiplicative'
        )

        # Modèles pour catastrophes
        self.disaster_cnn = self._build_disaster_cnn()
        self.weather_prophet = Prophet(
            changepoint_prior_scale=0.15,
            seasonality_mode='additive'
        )

        # Modèle de fusion pour priorités
        self.priority_model = RandomForestClassifier(n_estimators=100)

    def predict_dual_risk(self,
                          region_data: pd.DataFrame,
                          timeframe: str = '72h') -> Dict:
        """
        Prédit risques doubles pour une région
        """
        # 1. Prédictions conflits
        conflict_risk = self._predict_conflict_risk(region_data)
        conflict_timeline = self._predict_conflict_timeline(region_data)

        # 2. Prédictions catastrophes
        disaster_risk = self._predict_disaster_risk(region_data)
        disaster_timeline = self._predict_disaster_timeline(region_data)

        # 3. Analyse corrélations
```

```
        correlation = self._analyze_risk_correlation(
            conflict_risk,
            disaster_risk,
            region_data
        )

        # 4. Recommandations prioritaires
        priority_action = self._determine_priority_action(
            conflict_risk,
            disaster_risk,
            correlation
        )

        return {
            'conflict': {
                'probability': float(conflict_risk['probability']),
                'severity': conflict_risk['severity'],
                'timeline': conflict_timeline,
                'type': conflict_risk['conflict_type'],
                'indicators': conflict_risk['top_indicators']
            },
            'disaster': {
                'probability': float(disaster_risk['probability']),
                'type': disaster_risk['disaster_type'],
                'severity': disaster_risk['severity'],
                'timeline': disaster_timeline,
                'impact_area': disaster_risk['affected_area']
            },
            'correlation': {
                'score': correlation['score'],
                'interaction': correlation['interaction_type'],
                'compound_risk': correlation['compound_risk']
            },
            'recommendations': {
                'priority': priority_action['priority'],
                'actions': priority_action['actions'],
                'resources_needed': priority_action['resources'],
                'coordination': priority_action['coordination_needs']
            },
            'confidence': self._calculate_prediction_confidence(
                conflict_risk,
                disaster_risk
            )
        }

    def _build_conflict_lstm(self) -> tf.keras.Model:
        """LSTM pour séries temporelles conflits"""
```

```python
        model = tf.keras.Sequential([
            tf.keras.layers.LSTM(128, return_sequences=True, input_shape=(30, 15)),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.LSTM(64, return_sequences=True),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.LSTM(32),
            tf.keras.layers.Dense(16, activation='relu'),
            tf.keras.layers.Dense(4, activation='softmax')  # 4 niveaux de risque
        ])

        model.compile(
            optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy', tf.keras.metrics.AUC()]
        )

        return model

    def _build_disaster_cnn(self) -> tf.keras.Model:
        """CNN pour patterns catastrophes (images satellites + données)"""
        input_img = tf.keras.layers.Input(shape=(256, 256, 3))
        input_data = tf.keras.layers.Input(shape=(50,))

        # Branch CNN pour images
        x = tf.keras.layers.Conv2D(32, 3, activation='relu')(input_img)
        x = tf.keras.layers.MaxPooling2D(2)(x)
        x = tf.keras.layers.Conv2D(64, 3, activation='relu')(x)
        x = tf.keras.layers.MaxPooling2D(2)(x)
        x = tf.keras.layers.Conv2D(128, 3, activation='relu')(x)
        x = tf.keras.layers.GlobalAveragePooling2D()(x)

        # Branch Dense pour données
        y = tf.keras.layers.Dense(64, activation='relu')(input_data)
        y = tf.keras.layers.Dropout(0.3)(y)
        y = tf.keras.layers.Dense(32, activation='relu')(y)

        # Fusion
        combined = tf.keras.layers.concatenate([x, y])
        z = tf.keras.layers.Dense(64, activation='relu')(combined)
        z = tf.keras.layers.Dropout(0.3)(z)
        output = tf.keras.layers.Dense(5, activation='softmax')(z)  # 5 types catastro

        model = tf.keras.Model(inputs=[input_img, input_data], outputs=output)
        model.compile(
            optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy']
```

```python
        )

        return model

    def _predict_conflict_timeline(self, data: pd.DataFrame) -> pd.DataFrame:
        """Prophet pour timeline conflits"""
        # Préparer données pour Prophet
        prophet_data = pd.DataFrame({
            'ds': data['date'],
            'y': data['tension_index']
        })

        # Fit et prédiction
        self.social_prophet.fit(prophet_data)
        future = self.social_prophet.make_future_dataframe(periods=30, freq='D')
        forecast = self.social_prophet.predict(future)

        # Identifier points critiques
        critical_points = self._identify_critical_points(forecast)

        return {
            'forecast': forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail(30),
            'critical_dates': critical_points,
            'trend': 'escalating' if forecast['trend'].iloc[-1] > 0 else 'de-escalatin
        }

# Exemple utilisation
predictor = DualMissionPredictor()

# Données région avec risques multiples
region_data = pd.DataFrame({
    'date': pd.date_range('2025-01-01', periods=30, freq='D'),
    'tension_index': np.random.normal(0.6, 0.1, 30),
    'seismic_activity': np.random.normal(0.3, 0.05, 30),
    'weather_pressure': np.random.normal(1013, 5, 30),
    'social_media_sentiment': np.random.normal(-0.2, 0.3, 30)
})

predictions = predictor.predict_dual_risk(region_data)
print(f"Conflict risk: {predictions['conflict']['probability']:.2%}")
print(f"Disaster risk: {predictions['disaster']['probability']:.2%}")
print(f"Priority action: {predictions['recommendations']['priority']}")
```

🟠 PHOENIX : Intégration OpenCV + Edge AI

Installation

```bash
pip install opencv-python tensorflow-lite edge-tpu pillow
```

Système Réponse Rapide

python

```python
# phoenix/ai_integration/rapid_response.py
import cv2
import numpy as np
from typing import List, Dict, Tuple
import tensorflow as tf
from concurrent.futures import ThreadPoolExecutor
import time

class RapidResponseAI:
    """
    PHOENIX utilise Computer Vision pour urgences DOUBLE MISSION
    """

    def __init__(self):
        # Modèles optimisés pour edge
        self.crowd_detector = self._load_tflite_model('crowd_detection.tflite')
        self.damage_assessor = self._load_tflite_model('damage_assessment.tflite')
        self.evacuation_planner = self._load_tflite_model('evacuation_routes.tflite')

        # OpenCV pour traitement temps réel
        self.face_cascade = cv2.CascadeClassifier(
            cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'
        )

        # Thread pool pour parallélisation
        self.executor = ThreadPoolExecutor(max_workers=8)

    def analyze_emergency_scene(self,
                                video_feed: str,
                                emergency_type: str) -> Dict:
        """
        Analyse scène en temps réel pour double mission
        Target: <100ms par frame
        """
        start_time = time.time()

        # Capture vidéo
        cap = cv2.VideoCapture(video_feed)
        ret, frame = cap.read()

        if not ret:
            return {'error': 'Cannot read video feed'}

        # Analyses parallèles selon urgence
        if emergency_type == 'conflict':
            results = self._analyze_conflict_scene(frame)
```

```python
        elif emergency_type == 'disaster':
            results = self._analyze_disaster_scene(frame)
        else:  # Double urgence
            results = self._analyze_combined_emergency(frame)

        # Temps de traitement
        processing_time = (time.time() - start_time) * 1000

        cap.release()

        return {
            **results,
            'processing_time_ms': processing_time,
            'meets_latency_target': processing_time < 100
        }

    def _analyze_conflict_scene(self, frame: np.ndarray) -> Dict:
        """Analyse scène de conflit/manifestation"""
        # Détection foule
        crowd_density = self._detect_crowd_density(frame)

        # Analyse mouvement
        movement_patterns = self._analyze_movement_patterns(frame)

        # Détection tension
        tension_indicators = self._detect_tension_indicators(frame)

        # Plan de désescalade
        deescalation_plan = self._generate_deescalation_plan(
            crowd_density,
            movement_patterns,
            tension_indicators
        )

        return {
            'scene_type': 'conflict',
            'crowd_density': crowd_density,
            'tension_level': tension_indicators['level'],
            'movement_pattern': movement_patterns['pattern'],
            'recommended_actions': deescalation_plan,
            'safe_zones': self._identify_safe_zones(frame),
            'evacuation_routes': self._plan_peaceful_dispersal(frame)
        }

    def _analyze_disaster_scene(self, frame: np.ndarray) -> Dict:
        """Analyse scène de catastrophe"""
        # Évaluation dégâts
```

```python
        damage_assessment = self._assess_structural_damage(frame)

        # Détection victimes
        victims_detected = self._detect_victims(frame)

        # Routes évacuation
        evacuation_routes = self._calculate_evacuation_routes(frame)

        # Ressources nécessaires
        resources_needed = self._estimate_resources(
            damage_assessment,
            victims_detected
        )

        return {
            'scene_type': 'disaster',
            'damage_level': damage_assessment['severity'],
            'victims_count': len(victims_detected),
            'victim_locations': victims_detected,
            'evacuation_routes': evacuation_routes,
            'blocked_paths': damage_assessment['blocked_areas'],
            'resources_needed': resources_needed,
            'priority_zones': self._prioritize_rescue_zones(frame, victims_detected)
        }

    def _detect_crowd_density(self, frame: np.ndarray) -> Dict:
        """Utilise OpenCV + TFLite pour densité foule"""
        # Prétraitement
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        resized = cv2.resize(gray, (320, 240))

        # Détection visages pour estimation rapide
        faces = self.face_cascade.detectMultiScale(gray, 1.1, 4)

        # Modèle TFLite pour estimation précise
        input_data = np.expand_dims(resized, axis=0).astype(np.float32)
        self.crowd_detector.set_tensor(
            self.crowd_detector.get_input_details()[0]['index'],
            input_data
        )
        self.crowd_detector.invoke()

        density_map = self.crowd_detector.get_tensor(
            self.crowd_detector.get_output_details()[0]['index']
        )

        return {
```

```
            'estimated_count': len(faces) * 2.5,  # Estimation rapide
            'density_map': density_map,
            'risk_level': self._calculate_crowd_risk(density_map),
            'hotspots': self._identify_crowd_hotspots(density_map)
        }

    def _calculate_evacuation_routes(self, frame: np.ndarray) -> List[Dict]:
        """Calcul routes évacuation optimales"""
        # Edge detection pour obstacles
        edges = cv2.Canny(frame, 50, 150)

        # Pathfinding avec A*
        routes = []

        # Identifier sorties
        exits = self._detect_exits(frame)

        for exit_point in exits:
            route = {
                'exit_location': exit_point,
                'capacity': self._estimate_exit_capacity(frame, exit_point),
                'distance': self._calculate_average_distance(frame, exit_point),
                'obstacles': self._detect_obstacles_on_path(edges, exit_point),
                'estimated_time': self._estimate_evacuation_time(exit_point)
            }
            routes.append(route)

        # Trier par efficacité
        routes.sort(key=lambda x: x['estimated_time'])

        return routes[:5]  # Top 5 routes

# Utilisation edge deployment
responder = RapidResponseAI()

# Simulation urgence double (conflit pendant alerte cyclone)
emergency_response = responder.analyze_emergency_scene(
    video_feed="rtsp://camera.local/feed1",
    emergency_type="combined"
)

print(f"Temps de réponse: {emergency_response['processing_time_ms']:.1f}ms")
print(f"Actions prioritaires: {emergency_response['recommended_actions']}")
```

ATLAS : Intégration Scikit-learn + Folium

Installation

```bash
pip install scikit-learn folium geopandas networkx geopy
```

Système Mobilisation Intelligente

```
python
```

```python
# atlas/ai_integration/volunteer_mobilizer.py
from sklearn.cluster import KMeans, DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.metrics.pairwise import haversine_distances
import folium
import pandas as pd
import networkx as nx
from typing import List, Dict, Tuple

class IntelligentMobilizer:
    """
    ATLAS mobilise volontaires pour PAIX et CATASTROPHES
    """

    def __init__(self):
        self.volunteer_db = self._load_volunteer_database()
        self.skill_encoder = self._init_skill_encoder()
        self.location_clusterer = KMeans(n_clusters=50)
        self.network_analyzer = nx.Graph()

    def mobilize_for_crisis(self,
                            crisis: Dict,
                            target_count: int = 100) -> Dict:
        """
        Mobilise les bons volontaires pour la bonne crise
        """
        # 1. Identifier besoins selon type crise
        if crisis['type'] == 'conflict_mediation':
            required_skills = self._get_mediation_skills()
            volunteer_pool = self._filter_mediators()
        elif crisis['type'] == 'natural_disaster':
            required_skills = self._get_disaster_skills()
            volunteer_pool = self._filter_rescuers()
        else:  # Double crise
            required_skills = self._get_combined_skills()
            volunteer_pool = self.volunteer_db  # Tous disponibles

        # 2. Matching intelligent
        matched_volunteers = self._intelligent_matching(
            volunteer_pool,
            required_skills,
            crisis['location'],
            target_count
        )

        # 3. Optimisation déploiement
```

```
        deployment_plan = self._optimize_deployment(
            matched_volunteers,
            crisis
        )

        # 4. Génération carte interactive
        mobilization_map = self._create_mobilization_map(
            deployment_plan,
            crisis
        )

        return {
            'matched_volunteers': len(matched_volunteers),
            'deployment_plan': deployment_plan,
            'estimated_arrival_times': self._calculate_arrival_times(deployment_plan),
            'skill_coverage': self._assess_skill_coverage(matched_volunteers, required
            'mobilization_map': mobilization_map,
            'coordination_groups': self._form_coordination_groups(matched_volunteers),
            'backup_volunteers': self._identify_backups(volunteer_pool, matched_volunt
        }

    def _intelligent_matching(self,
                              volunteers: pd.DataFrame,
                              required_skills: List[str],
                              crisis_location: Tuple[float, float],
                              target_count: int) -> pd.DataFrame:
        """
        Matching ML multi-critères
        """
        # Calculer scores pour chaque volontaire
        volunteers['match_score'] = volunteers.apply(
            lambda v: self._calculate_match_score(v, required_skills, crisis_location)
            axis=1
        )

        # Features pour clustering
        features = []
        for _, volunteer in volunteers.iterrows():
            features.append([
                volunteer['match_score'],
                self._calculate_distance(volunteer['location'], crisis_location),
                volunteer['experience_score'],
                volunteer['availability_score'],
                self._encode_skills(volunteer['skills'])
            ])

        # Normalisation
```

```python
        scaler = StandardScaler()
        features_scaled = scaler.fit_transform(features)

        # Clustering pour groupes cohérents
        clusters = DBSCAN(eps=0.3, min_samples=5).fit_predict(features_scaled)
        volunteers['cluster'] = clusters

        # Sélection optimale par cluster
        selected = pd.DataFrame()

        for cluster_id in set(clusters):
            if cluster_id != -1:  # Ignorer noise
                cluster_volunteers = volunteers[volunteers['cluster'] == cluster_id]

                # Prendre les meilleurs de chaque cluster
                n_from_cluster = min(
                    len(cluster_volunteers),
                    max(1, int(target_count * len(cluster_volunteers) / len(volunteers
                )

                best_from_cluster = cluster_volunteers.nlargest(n_from_cluster, 'match
                selected = pd.concat([selected, best_from_cluster])

        # Compléter si nécessaire
        if len(selected) < target_count:
            remaining = volunteers[~volunteers.index.isin(selected.index)]
            additional = remaining.nlargest(target_count - len(selected), 'match_score
            selected = pd.concat([selected, additional])

        return selected.head(target_count)

    def _optimize_deployment(self,
                              volunteers: pd.DataFrame,
                              crisis: Dict) -> List[Dict]:
        """
        Optimise déploiement avec graphes
        """
        # Créer graphe de déploiement
        G = nx.Graph()

        # Ajouter nœuds (volontaires + points de crise)
        for idx, volunteer in volunteers.iterrows():
            G.add_node(f"v_{idx}",
                      type='volunteer',
                      location=volunteer['location'],
                      skills=volunteer['skills'])
```

```
        # Ajouter points d'intervention
        for i, point in enumerate(crisis['intervention_points']):
            G.add_node(f"p_{i}",
                      type='point',
                      location=point['location'],
                      priority=point['priority'])

        # Calculer distances et créer arêtes
        for v_node in [n for n in G.nodes() if n.startswith('v_')]:
            for p_node in [n for n in G.nodes() if n.startswith('p_')]:
                distance = self._calculate_distance(
                    G.nodes[v_node]['location'],
                    G.nodes[p_node]['location']
                )
                G.add_edge(v_node, p_node, weight=distance)

        # Résoudre affectation optimale
        deployment = []

        # Utiliser algorithme hongrois pour affectation optimale
        from scipy.optimize import linear_sum_assignment

        # Matrice de coûts
        cost_matrix = nx.to_numpy_array(G)
        row_ind, col_ind = linear_sum_assignment(cost_matrix)

        for v_idx, p_idx in zip(row_ind, col_ind):
            if v_idx < len(volunteers) and p_idx < len(crisis['intervention_points']):
                deployment.append({
                    'volunteer_id': volunteers.iloc[v_idx]['id'],
                    'assigned_point': crisis['intervention_points'][p_idx],
                    'estimated_distance': cost_matrix[v_idx, p_idx],
                    'skills_match': self._calculate_point_skill_match(
                        volunteers.iloc[v_idx]['skills'],
                        crisis['intervention_points'][p_idx]['needs']
                    )
                })

        return deployment

    def _create_mobilization_map(self,
                                deployment_plan: List[Dict],
                                crisis: Dict) -> folium.Map:
        """
        Carte interactive de mobilisation
        """
        # Centrer sur zone de crise
```

```python
crisis_center = crisis['location']
m = folium.Map(location=crisis_center, zoom_start=10)

# Ajouter marqueurs volontaires
for deployment in deployment_plan:
    volunteer = self.volunteer_db[
        self.volunteer_db['id'] == deployment['volunteer_id']
    ].iloc[0]

    # Couleur selon type mission
    if 'mediation' in volunteer['skills']:
        color = 'blue'  # Paix
    elif 'first_aid' in volunteer['skills']:
        color = 'red'   # Urgence
    else:
        color = 'green' # Polyvalent

    folium.Marker(
        volunteer['location'],
        popup=f"Volunteer: {volunteer['name']}<br>Skills: {volunteer['skills']
        icon=folium.Icon(color=color, icon='user')
    ).add_to(m)

    # Ligne vers point d'affectation
    folium.PolyLine(
        [volunteer['location'], deployment['assigned_point']['location']],
        color=color,
        weight=2,
        opacity=0.8
    ).add_to(m)

# Ajouter zones d'intervention
for point in crisis['intervention_points']:
    folium.CircleMarker(
        point['location'],
        radius=point['priority'] * 10,
        popup=f"Priority: {point['priority']}<br>Needs: {point['needs']}",
        color='red',
        fill=True
    ).add_to(m)

# Zone de danger/conflit
if crisis['affected_area']:
    folium.Polygon(
        crisis['affected_area'],
        color='orange',
        fill=True,
```

```python
                fillOpacity=0.2
            ).add_to(m)

        return m

# Exemple utilisation
mobilizer = IntelligentMobilizer()

# Double crise : tensions + inondations
crisis_scenario = {
    'type': 'combined',
    'location': (12.9716, 77.5946),  # Bangalore
    'intervention_points': [
        {
            'location': (12.9716, 77.5946),
            'priority': 5,
            'needs': ['mediation', 'translation']
        },
        {
            'location': (12.9816, 77.5846),
            'priority': 4,
            'needs': ['first_aid', 'evacuation']
        }
    ],
    'affected_area': [(12.96, 77.58), (12.98, 77.58), (12.98, 77.60), (12.96, 77.60)]
}

mobilization = mobilizer.mobilize_for_crisis(crisis_scenario, target_count=50)
print(f"Mobilized {mobilization['matched_volunteers']} volunteers")
print(f"Skill coverage: {mobilization['skill_coverage']}")

# Sauver carte
mobilization['mobilization_map'].save('crisis_mobilization.html')
```

📊 Tableau Récapitulatif des Intégrations

Système	IA Principale	Mission Paix	Mission Catastrophe	Priorité
ETHOS	SHAP + Fairlearn	Validation éthique médiation	Validation éthique évacuation	🔴 MAX
NEXUS	Transformers	Traduction négociations	Instructions multilingues	🟡 HAUTE
CHRONOS	TensorFlow + Prophet	Prédire tensions	Prédire séismes/météo	🟡 HAUTE
PHOENIX	OpenCV + TFLite	Analyser foules	Détecter victimes	🟠 MOYENNE
ATLAS	Scikit-learn	Matcher médiateurs	Mobiliser secouristes	🟢 NORMALE

🔑 Points Clés pour les Développeurs

1. Toujours Penser Double Mission

Chaque fonction doit gérer les DEUX cas d'usage

2. Performance Critique

- PHOENIX : <100ms obligatoire
- CHRONOS : Prédictions en temps réel
- NEXUS : Réponses <500ms

3. Éthique Intégrée

Chaque décision IA passe par ETHOS

4. Scalabilité Massive

Prévoir 1M+ utilisateurs simultanés en crise

5. Tests Double Scénario

Toujours tester paix ET catastrophe

Ce guide sera mis à jour avec chaque nouvelle intégration. Contribuez vos exemples !