# Core Miner Implementation Plan

## Alert Scoring, Prioritization, and Cluster Assessment (Proposals 1, 2, 3)

**Date**: 2025-10-25
**Purpose**: Complete implementation guide for miners
**Scope**: Proposals 1, 2, 3 - Core miner ML capabilities

---

## Overview

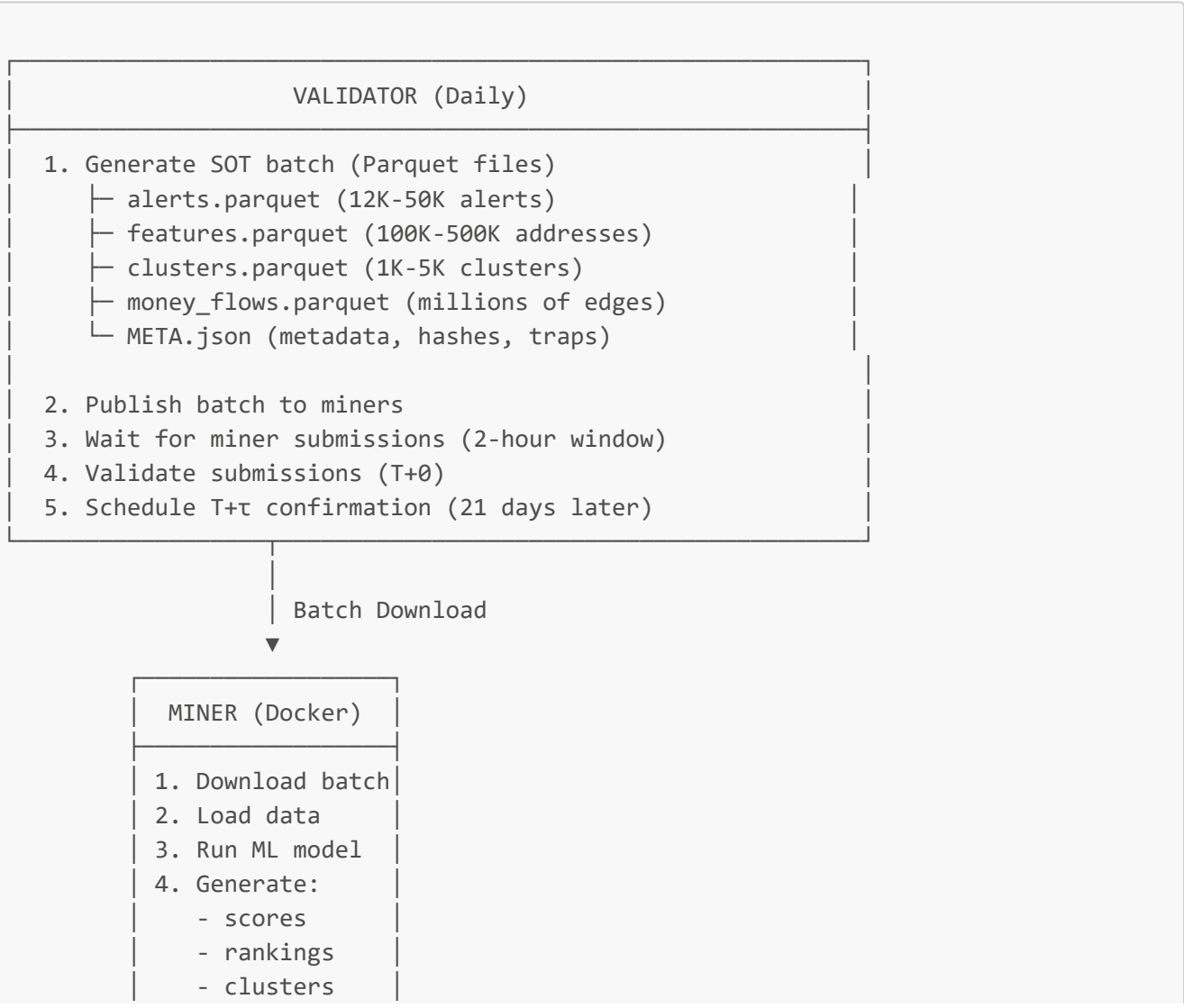
This document provides the complete implementation plan for core miner capabilities:

- **Proposal 1**: Alert Risk Scoring - Score each alert with P(illicit within τ days)
- **Proposal 2**: Alert Prioritization - Rank alerts by investigation urgency
- **Proposal 3**: Cluster Risk Assessment - Score alert clusters

All implemented using **deterministic ML models** executed in Docker containers.

---

## Architecture Overview



```
          │ 5. Sign & submit │
          └──────────┬───────┘
                     │
                     │ Submission Upload
                     ▼
          ┌──────────────────────┐
          │  VALIDATOR           │
          │  (Validation Layer)  │
          ├──────────────────────┤
          │ T+0: Immediate       │
          │ ├─ Format checks     │
          │ ├─ Pattern traps     │
          │ └─ Sanity tests      │
          │                      │
          │ T+τ: Ground Truth    │
          │ ├─ Outcome tracking  │
          │ ├─ Metric computation│
          │ └─ Reward distribution│
          └──────────────────────┘
```

---

# Proposal 1: Alert Risk Scoring

## 1.1 Training Phase (Miner-Side)

**File:** `miner/models/alert_scorer.py`

```python
"""
Alert Risk Scoring Model
Predicts P(illicit within τ days) for each alert
"""

import numpy as np
import pandas as pd
import lightgbm as lgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, brier_score_loss, log_loss
from loguru import logger
import json

class AlertScorerModel:
    """
    LightGBM model for alert risk scoring
    """

    def __init__(self, seed=42):
        self.seed = seed
        self.model = None
        self.feature_names = None
        self.model_version = None
```

```
    def prepare_features(
        self,
        alerts: pd.DataFrame,
        features: pd.DataFrame,
        clusters: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Prepare feature matrix for training/prediction

        Combines:
        - Alert metadata (typology_type, confidence, severity)
        - Address features (140+ features per address)
        - Cluster features (if alert is in cluster)
        """
        # Start with alert data
        X = alerts[['alert_id', 'address']].copy()

        # Add alert-level features
        X['alert_confidence_score'] = alerts['alert_confidence_score']
        X['severity_encoded'] = alerts['severity'].map({
            'low': 0.25, 'medium': 0.5, 'high': 0.75, 'critical': 1.0
        })
        X['volume_usd'] = alerts['volume_usd']

        # One-hot encode typology_type
        typology_dummies = pd.get_dummies(
            alerts['typology_type'],
            prefix='typology'
        )
        X = pd.concat([X, typology_dummies], axis=1)

        # Merge address features
        X = X.merge(
            features,
            left_on='address',
            right_on='address',
            how='left'
        )

        # Merge cluster features (if available)
        alert_clusters =
alerts[alerts['alert_id'].isin(clusters['related_alert_ids'].explode())]
        if len(alert_clusters) > 0:
            cluster_features = self._extract_cluster_features(clusters)
            X = X.merge(cluster_features, on='alert_id', how='left')
            X['in_cluster'] = X['cluster_id'].notna().astype(int)
        else:
            X['in_cluster'] = 0

        # Fill NaN with 0
        X = X.fillna(0)

        # Store feature names (exclude id columns)
        self.feature_names = [
```

```
            col for col in X.columns
            if col not in ['alert_id', 'address']
        ]

        return X

    def _extract_cluster_features(self, clusters: pd.DataFrame) -> pd.DataFrame:
        """Extract features from cluster data"""
        cluster_features = []

        for _, cluster in clusters.iterrows():
            for alert_id in cluster['related_alert_ids']:
                cluster_features.append({
                    'alert_id': alert_id,
                    'cluster_id': cluster['cluster_id'],
                    'cluster_size': cluster['total_alerts'],
                    'cluster_volume_usd': cluster['total_volume_usd'],
                    'cluster_severity_max':
self._encode_severity(cluster['severity_max']),
                    'cluster_confidence_avg': cluster['confidence_avg']
                })

        return pd.DataFrame(cluster_features)

    def _encode_severity(self, severity: str) -> float:
        """Encode severity to numeric"""
        mapping = {'low': 0.25, 'medium': 0.5, 'high': 0.75, 'critical': 1.0}
        return mapping.get(severity, 0.5)

    def train(
        self,
        X_train: pd.DataFrame,
        y_train: np.ndarray,
        X_val: pd.DataFrame = None,
        y_val: np.ndarray = None
    ):
        """
        Train LightGBM model on historical data with confirmed outcomes

        Args:
            X_train: Training features
            y_train: Binary outcomes (1=confirmed illicit, 0=not illicit)
            X_val: Validation features (optional)
            y_val: Validation outcomes (optional)
        """
        logger.info(f"Training alert scorer on {len(X_train)} samples")

        # Prepare feature matrix
        feature_matrix = X_train[self.feature_names]

        # LightGBM parameters
        params = {
            'objective': 'binary',
            'metric': ['auc', 'binary_logloss'],
```

```
            'boosting_type': 'gbdt',
            'num_leaves': 31,
            'learning_rate': 0.05,
            'feature_fraction': 0.8,
            'bagging_fraction': 0.8,
            'bagging_freq': 5,
            'verbose': -1,
            'seed': self.seed,
            'deterministic': True
        }

        # Create dataset
        train_data = lgb.Dataset(
            feature_matrix,
            label=y_train,
            feature_name=self.feature_names
        )

        # Validation set
        valid_sets = [train_data]
        valid_names = ['train']

        if X_val is not None and y_val is not None:
            val_data = lgb.Dataset(
                X_val[self.feature_names],
                label=y_val,
                reference=train_data
            )
            valid_sets.append(val_data)
            valid_names.append('valid')

        # Train model
        self.model = lgb.train(
            params,
            train_data,
            num_boost_round=500,
            valid_sets=valid_sets,
            valid_names=valid_names,
            callbacks=[
                lgb.early_stopping(stopping_rounds=50),
                lgb.log_evaluation(period=50)
            ]
        )

        # Evaluate
        if X_val is not None and y_val is not None:
            val_pred = self.model.predict(X_val[self.feature_names])
            auc = roc_auc_score(y_val, val_pred)
            brier = brier_score_loss(y_val, val_pred)
            logloss = log_loss(y_val, val_pred)

            logger.info(f"Validation metrics: AUC={auc:.4f}, Brier={brier:.4f},
LogLoss={logloss:.4f}")
```

```
        # Set model version (hash of model file)
        self.model_version = self._compute_model_hash()
        logger.info(f"Model trained, version={self.model_version}")

    def predict(self, X: pd.DataFrame) -> np.ndarray:
        """
        Predict risk scores for alerts

        Returns:
            Array of probabilities in [0, 1]
        """
        if self.model is None:
            raise ValueError("Model not trained. Call train() first.")

        feature_matrix = X[self.feature_names]
        scores = self.model.predict(feature_matrix)

        # Ensure scores are in [0, 1]
        scores = np.clip(scores, 0, 1)

        return scores

    def save_model(self, path: str):
        """Save model to file"""
        self.model.save_model(path)

        # Save metadata
        metadata = {
            'model_version': self.model_version,
            'feature_names': self.feature_names,
            'num_features': len(self.feature_names),
            'seed': self.seed
        }

        with open(path + '.meta', 'w') as f:
            json.dump(metadata, f, indent=2)

        logger.info(f"Model saved to {path}")

    def load_model(self, path: str):
        """Load model from file"""
        self.model = lgb.Booster(model_file=path)

        # Load metadata
        with open(path + '.meta', 'r') as f:
            metadata = json.load(f)

        self.feature_names = metadata['feature_names']
        self.model_version = metadata['model_version']
        self.seed = metadata['seed']

        logger.info(f"Model loaded from {path}, version={self.model_version}")

    def _compute_model_hash(self) -> str:
```

```
        """Compute hash of model for versioning"""
        import hashlib
        model_str = self.model.model_to_string()
        return hashlib.sha256(model_str.encode()).hexdigest()[:12]
```

**Training Script:** `miner/scripts/train_alert_scorer.py`

```
"""
Train alert scorer model on historical data
"""

import pandas as pd
import numpy as np
from miner.models.alert_scorer import AlertScorerModel
from miner.data.historical_loader import HistoricalDataLoader
from loguru import logger
import argparse
from sklearn.model_selection import train_test_split

def load_training_data(start_date: str, end_date: str):
    """
    Load historical alerts with confirmed outcomes
    """
    loader = HistoricalDataLoader()

    # Load historical batches
    historical_data = loader.load_date_range(start_date, end_date)

    # Filter for alerts with confirmed outcomes (T+τ passed)
    confirmed_alerts = historical_data[
        historical_data['confirmation_status'].notna()
    ]

    logger.info(f"Loaded {len(confirmed_alerts)} confirmed alerts")

    # Prepare features and labels
    alerts = confirmed_alerts[['alert_id', 'address', 'typology_type',
                               'alert_confidence_score', 'severity', 'volume_usd']]

    features = confirmed_alerts[['address'] + FEATURE_COLUMNS]
    clusters = loader.load_clusters(start_date, end_date)

    # Binary labels: 1 = confirmed illicit, 0 = not illicit
    labels = (confirmed_alerts['confirmation_status'] ==
'confirmed_illicit').astype(int)

    return alerts, features, clusters, labels

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--start-date', required=True)
```

```
    parser.add_argument('--end-date', required=True)
    parser.add_argument('--output', default='models/alert_scorer_v1.txt')
    args = parser.parse_args()

    # Load data
    logger.info("Loading training data...")
    alerts, features, clusters, labels = load_training_data(
        args.start_date, args.end_date
    )

    # Initialize model
    model = AlertScorerModel(seed=42)

    # Prepare features
    logger.info("Preparing features...")
    X = model.prepare_features(alerts, features, clusters)

    # Split train/val
    X_train, X_val, y_train, y_val = train_test_split(
        X, labels, test_size=0.2, random_state=42, stratify=labels
    )

    # Train
    logger.info("Training model...")
    model.train(X_train, y_train, X_val, y_val)

    # Save
    model.save_model(args.output)
    logger.info(f"Model saved to {args.output}")

if __name__ == '__main__':
    main()
```

---

## 1.2 Inference Phase (Production Scoring)

**File:** `miner/miner_main.py`

```
"""
Main miner entry point for daily scoring
Runs in Docker container with no network access
"""

import argparse
import json
import time
import pandas as pd
import numpy as np
from pathlib import Path
from loguru import logger
import hashlib
```

```
from miner.models.alert_scorer import AlertScorerModel
from miner.models.alert_ranker import AlertRankerModel
from miner.models.cluster_scorer import ClusterScorerModel

RANDOM_SEED = 42

def verify_input_integrity(input_dir: Path):
    """
    Verify input data integrity using META.json
    """
    meta_path = input_dir / "META.json"
    with open(meta_path) as f:
        meta = json.load(f)

    # Verify file hashes
    for filename, expected_hash in meta['sha256'].items():
        file_path = input_dir / filename
        computed_hash = compute_file_hash(file_path)

        if computed_hash != expected_hash:
            raise ValueError(f"Hash mismatch for {filename}")

    logger.info("✓ Input integrity verified")
    return meta

def compute_file_hash(path: Path) -> str:
    """Compute SHA256 hash of file"""
    sha256 = hashlib.sha256()
    with open(path, 'rb') as f:
        for chunk in iter(lambda: f.read(4096), b''):
            sha256.update(chunk)
    return sha256.hexdigest()

def load_inputs(input_dir: Path):
    """Load all input Parquet files"""
    alerts = pd.read_parquet(input_dir / "alerts.parquet")
    features = pd.read_parquet(input_dir / "features.parquet")
    clusters = pd.read_parquet(input_dir / "clusters.parquet")
    money_flows = pd.read_parquet(input_dir / "money_flows.parquet")

    logger.info(f"Loaded {len(alerts)} alerts, {len(features)} features,
{len(clusters)} clusters")

    return alerts, features, clusters, money_flows

def score_alerts(alerts, features, clusters, model_path: str):
    """
    Score all alerts using trained model
    """
    logger.info("Loading alert scorer model...")
    model = AlertScorerModel()
    model.load_model(model_path)
```

```
    # Prepare features
    logger.info("Preparing features...")
    X = model.prepare_features(alerts, features, clusters)

    # Predict
    logger.info("Computing scores...")
    scores = model.predict(X)

    # Create output dataframe
    output = pd.DataFrame({
        'alert_id': alerts['alert_id'],
        'score': scores,
        'model_version': model.model_version,
        'explain_json': create_explanations(X, model, scores)
    })

    return output

def create_explanations(X, model, scores):
    """
    Create explanation JSON for each prediction
    Uses SHAP values (top 5 features)
    """
    import shap

    # Compute SHAP values
    explainer = shap.TreeExplainer(model.model)
    shap_values = explainer.shap_values(X[model.feature_names])

    explanations = []
    for i, score in enumerate(scores):
        # Get top 5 features by absolute SHAP value
        feature_impacts = list(zip(
            model.feature_names,
            shap_values[i]
        ))
        top_features = sorted(
            feature_impacts,
            key=lambda x: abs(x[1]),
            reverse=True
        )[:5]

        explanation = {
            'score': float(score),
            'top_features': [
                {'feature': f, 'impact': float(v)}
                for f, v in top_features
            ]
        }

        explanations.append(json.dumps(explanation))

    return explanations
```

```python
def create_manifest(model_version: str):
    """Create manifest.json"""
    return {
        'repo_commit': 'latest',  # Git commit hash
        'image_digest': 'sha256:...',  # Docker image digest
        'model_version': model_version,
        'libs': {
            'python': '3.11',
            'lightgbm': '4.3.0',
            'numpy': '1.26.0',
            'pandas': '2.1.0'
        },
        'seeds': {'global': RANDOM_SEED},
        'hardware': {'cpu': '4 vCPU', 'ram_gb': 8}
    }

def create_receipt(input_hash: str, image_digest: str, scores_hash: str,
model_version: str):
    """Create signed receipt"""
    import nacl.signing
    import nacl.encoding

    # Load signing key (from environment or file)
    signing_key =
nacl.signing.SigningKey.from_seed(bytes.fromhex(MINER_PRIVATE_KEY))

    receipt_data = {
        'input_hash': input_hash,
        'image_digest': image_digest,
        'scores_hash': scores_hash,
        'model_version': model_version,
        'timestamp': pd.Timestamp.now().isoformat(),
        'miner_pubkey':
signing_key.verify_key.encode(encoder=nacl.encoding.HexEncoder).decode()
    }

    # Sign
    message = json.dumps(receipt_data, sort_keys=True).encode()
    signed = signing_key.sign(message)

    receipt_data['signature'] = signed.signature.hex()

    return receipt_data

def main(input_dir: str, output_dir: str, model_path: str =
'models/alert_scorer_v1.txt'):
    """
    Main miner execution
    """
    t0 = time.time()

    # Setup
    input_path = Path(input_dir)
    output_path = Path(output_dir)
```

```
    output_path.mkdir(parents=True, exist_ok=True)

    # Verify inputs
    meta = verify_input_integrity(input_path)

    # Load data
    alerts, features, clusters, money_flows = load_inputs(input_path)

    # Score alerts (Proposal 1)
    scores_df = score_alerts(alerts, features, clusters, model_path)

    # Compute latency
    elapsed_ms = int((time.time() - t0) * 1000)
    latency_per_alert = elapsed_ms // max(1, len(alerts))
    scores_df['latency_ms'] = latency_per_alert

    # Save scores
    scores_path = output_path / "scores.parquet"
    scores_df.to_parquet(scores_path, index=False)
    logger.info(f"Scores saved to {scores_path}")

    # Create manifest
    manifest = create_manifest(scores_df['model_version'].iloc[0])
    with open(output_path / "manifest.json", 'w') as f:
        json.dump(manifest, f, indent=2)

    # Create receipt
    input_hash = compute_file_hash(input_path / "alerts.parquet")
    scores_hash = compute_file_hash(scores_path)
    receipt = create_receipt(
        input_hash,
        manifest['image_digest'],
        scores_hash,
        manifest['model_version']
    )
    with open(output_path / "receipt.json", 'w') as f:
        json.dump(receipt, f, indent=2)

    logger.info(f"✓ Scoring complete in {elapsed_ms}ms
({latency_per_alert}ms/alert)")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--input-dir", required=True)
    parser.add_argument("--output-dir", required=True)
    parser.add_argument("--model-path", default="models/alert_scorer_v1.txt")
    args = parser.parse_args()

    np.random.seed(RANDOM_SEED)
    main(args.input_dir, args.output_dir, args.model_path)
```

---

### 1.3 Docker Container

**File:** `miner/Dockerfile`

```
FROM python:3.11-slim

# Install dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    tini ca-certificates && \
    rm -rf /var/lib/apt/lists/*

# Set determinism environment variables
ENV PYTHONHASHSEED=0
ENV NUMEXPR_MAX_THREADS=4
ENV OMP_NUM_THREADS=4
ENV MKL_NUM_THREADS=4
ENV PYTHONUNBUFFERED=1

# Install Python packages
COPY requirements.txt /app/requirements.txt
RUN pip install --no-cache-dir -r /app/requirements.txt

# Copy miner code
COPY miner/ /app/miner/
COPY models/ /app/models/

WORKDIR /app

# Use tini for proper signal handling
ENTRYPOINT ["/usr/bin/tini", "--"]

# Default command
CMD ["python", "-u", "-m", "miner.miner_main", \
     "--input-dir", "/input", \
     "--output-dir", "/output", \
     "--model-path", "/app/models/alert_scorer_v1.txt"]
```

**File:** `miner/requirements.txt`

```
numpy==1.26.0
pandas==2.1.0
pyarrow==14.0.0
lightgbm==4.3.0
scikit-learn==1.3.0
shap==0.44.0
loguru==0.7.2
pynacl==1.5.0
```

**Run Command:**

```
# Build image
docker build -t miner-alert-scorer:v1 .

# Run scoring (no network access)
docker run --rm --network=none \
  -v /path/to/batch:/input:ro \
  -v /path/to/output:/output \
  miner-alert-scorer:v1
```

---

# Proposal 2: Alert Prioritization

## 2.1 Ranking Model

**File:** `miner/models/alert_ranker.py`

```
"""
Alert Prioritization using Learning-to-Rank
"""

import numpy as np
import pandas as pd
import lightgbm as lgb
from loguru import logger

class AlertRankerModel:
    """
    LambdaMART model for ranking alerts by urgency
    """

    def __init__(self, seed=42):
        self.seed = seed
        self.model = None
        self.feature_names = None

    def prepare_ranking_features(
        self,
        alerts: pd.DataFrame,
        features: pd.DataFrame,
        scores: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Prepare features for ranking
        Includes all alert features + predicted risk scores
        """
        # Merge with scores from Proposal 1
        X = alerts.merge(scores[['alert_id', 'score']], on='alert_id')
```

```
        # Add address features
        X = X.merge(features, on='address', how='left')

        # Additional ranking-specific features
        X['urgency_score'] = (
            X['score'] * 0.5 +  # Risk score
            X['severity_encoded'] * 0.3 +  # Severity
            X['alert_confidence_score'] * 0.2  # Confidence
        )

        X = X.fillna(0)

        self.feature_names = [
            col for col in X.columns
            if col not in ['alert_id', 'address']
        ]

        return X

    def train(self, X_train: pd.DataFrame, relevance: np.ndarray):
        """
        Train LambdaMART ranker

        Args:
            X_train: Features
            relevance: Relevance scores (e.g., 2=confirmed illicit, 1=suspicious,
0=clean)
        """
        logger.info(f"Training alert ranker on {len(X_train)} samples")

        params = {
            'objective': 'lambdarank',
            'metric': 'ndcg',
            'ndcg_eval_at': [10, 50, 100, 500],
            'num_leaves': 31,
            'learning_rate': 0.05,
            'seed': self.seed,
            'deterministic': True
        }

        train_data = lgb.Dataset(
            X_train[self.feature_names],
            label=relevance,
            group=[len(X_train)]  # Single query group
        )

        self.model = lgb.train(
            params,
            train_data,
            num_boost_round=300
        )

        logger.info("Ranker trained")
```

```
    def rank(self, X: pd.DataFrame) -> np.ndarray:
        """
        Predict ranking scores
        Higher scores = higher priority
        """
        ranking_scores = self.model.predict(X[self.feature_names])
        return ranking_scores
```

### 2.2 Integration in Miner

```
# In miner_main.py

def rank_alerts(alerts, features, scores_df, model_path: str):
    """
    Rank alerts by investigation priority
    """
    logger.info("Loading alert ranker model...")
    ranker = AlertRankerModel()
    ranker.load_model(model_path)

    # Prepare features
    X = ranker.prepare_ranking_features(alerts, features, scores_df)

    # Get ranking scores
    ranking_scores = ranker.rank(X)

    # Create ranked output
    ranked = pd.DataFrame({
        'alert_id': alerts['alert_id'],
        'ranking_score': ranking_scores,
        'rank': pd.Series(ranking_scores).rank(ascending=False,
method='min').astype(int)
    })

    return ranked.sort_values('rank')
```

---

## Proposal 3: Cluster Risk Assessment

### 3.1 Cluster Scoring Model

**File:** `miner/models/cluster_scorer.py`

```
"""
Cluster Risk Assessment Model
"""

import numpy as np
import pandas as pd
```

```python
import lightgbm as lgb
from loguru import logger

class ClusterScorerModel:
    """
    Score entire alert clusters
    """

    def __init__(self, seed=42):
        self.seed = seed
        self.model = None

    def prepare_cluster_features(
        self,
        clusters: pd.DataFrame,
        alerts: pd.DataFrame,
        features: pd.DataFrame,
        alert_scores: pd.DataFrame
    ) -> pd.DataFrame:
        """
        Prepare cluster-level features
        """
        cluster_features = []

        for _, cluster in clusters.iterrows():
            # Get alerts in this cluster
            cluster_alerts = alerts[
                alerts['alert_id'].isin(cluster['related_alert_ids'])
            ]

            # Get addresses in cluster
            cluster_addresses = features[
                features['address'].isin(cluster['addresses_involved'])
            ]

            # Get scores for cluster alerts
            cluster_scores = alert_scores[
                alert_scores['alert_id'].isin(cluster['related_alert_ids'])
            ]

            # Aggregate features
            cluster_feat = {
                'cluster_id': cluster['cluster_id'],

                # Basic cluster metrics
                'total_alerts': cluster['total_alerts'],
                'total_volume_usd': cluster['total_volume_usd'],
                'address_count': len(cluster['addresses_involved']),

                # Alert score statistics
                'alert_score_mean': cluster_scores['score'].mean(),
                'alert_score_max': cluster_scores['score'].max(),
                'alert_score_std': cluster_scores['score'].std(),
```

```
                # Address feature aggregates
                'avg_behavioral_anomaly':
cluster_addresses['behavioral_anomaly_score'].mean(),
                'avg_graph_anomaly':
cluster_addresses['graph_anomaly_score'].mean(),
                'avg_degree': cluster_addresses['degree_total'].mean(),
                'total_volume_all_addresses':
cluster_addresses['total_volume_usd'].sum(),

                # Temporal features
                'time_span_days': (cluster['latest_alert_timestamp'] -
                                   cluster['earliest_alert_timestamp']) / 86400000,

                # Cluster coherence
                'confidence_variance': cluster_addresses['quality_score'].std(),

                # Cluster type encoding
                'is_same_entity': int(cluster['cluster_type'] == 'same_entity'),
                'is_pattern': int(cluster['cluster_type'] == 'pattern'),
                'is_proximity': int(cluster['cluster_type'] == 'proximity')
            }

            cluster_features.append(cluster_feat)

        X = pd.DataFrame(cluster_features)
        X = X.fillna(0)

        return X

    def train(self, X_train: pd.DataFrame, y_train: np.ndarray):
        """
        Train cluster scorer

        Args:
            y_train: Binary (1=cluster with ≥1 confirmed illicit, 0=clean cluster)
        """
        logger.info(f"Training cluster scorer on {len(X_train)} clusters")

        params = {
            'objective': 'binary',
            'metric': 'auc',
            'num_leaves': 31,
            'learning_rate': 0.05,
            'seed': self.seed,
            'deterministic': True
        }

        feature_cols = [c for c in X_train.columns if c != 'cluster_id']

        train_data = lgb.Dataset(X_train[feature_cols], label=y_train)

        self.model = lgb.train(params, train_data, num_boost_round=300)

        logger.info("Cluster scorer trained")
```

```
    def score_clusters(self, X: pd.DataFrame) -> np.ndarray:
        """
        Predict cluster risk scores
        """
        feature_cols = [c for c in X.columns if c != 'cluster_id']
        scores = self.model.predict(X[feature_cols])
        return np.clip(scores, 0, 1)
```

---

## Validation Framework

### Validator-Side Validation

**File:** `validator/validation/miner_validator.py`

```
"""
Validator-side validation of miner submissions
"""

import pandas as pd
import numpy as np
from loguru import logger
import hashlib
import json

class MinerSubmissionValidator:
    """
    Validates miner submissions across 3 tiers
    """

    def validate_submission(
        self,
        miner_id: str,
        submission_path: str,
        batch_meta: dict,
        pattern_traps: list
    ) -> dict:
        """
        Full validation pipeline
        """
        validation_results = {
            'miner_id': miner_id,
            'timestamp': pd.Timestamp.now().isoformat(),
            'tier1_integrity': {},
            'tier2_pattern_traps': {},
            'tier3_pending': True  # Will be filled at T+τ
        }

        # Tier 1: Integrity checks
        validation_results['tier1_integrity'] = self._tier1_integrity_checks(
```

```
            submission_path, batch_meta
        )

        if not validation_results['tier1_integrity']['passed']:
            logger.warning(f"Miner {miner_id} failed Tier 1")
            return validation_results

        # Tier 2: Pattern trap checks
        validation_results['tier2_pattern_traps'] = self._tier2_pattern_traps(
            submission_path, pattern_traps
        )

        # Compute immediate score (Tier 1 + Tier 2)
        validation_results['immediate_score'] = self._compute_immediate_score(
            validation_results
        )

        return validation_results

    def _tier1_integrity_checks(self, submission_path: str, batch_meta: dict) ->
dict:
        """
        Tier 1: Format, hash, range, variance checks
        """
        scores = pd.read_parquet(f"{submission_path}/scores.parquet")
        manifest = json.load(open(f"{submission_path}/manifest.json"))
        receipt = json.load(open(f"{submission_path}/receipt.json"))

        checks = {}

        # Check 1: Format validation
        required_cols = ['alert_id', 'score', 'model_version', 'latency_ms']
        checks['format'] = all(col in scores.columns for col in required_cols)

        # Check 2: Range validation
        checks['range'] = scores['score'].between(0, 1).all()

        # Check 3: Variance check (not all same value)
        checks['variance'] = scores['score'].std() > 1e-6

        # Check 4: Hash verification
        scores_hash = self._compute_parquet_hash(f"
{submission_path}/scores.parquet")
        checks['hash'] = scores_hash == receipt['scores_hash']

        # Check 5: Signature verification
        checks['signature'] = self._verify_signature(receipt)

        # Check 6: Cardinality (one score per alert)
        expected_alerts = batch_meta['n_alerts']
        checks['cardinality'] = len(scores) == expected_alerts

        checks['passed'] = all(checks.values())
```

```
        return checks

    def _tier2_pattern_traps(self, submission_path: str, pattern_traps: list) ->
dict:
        """
        Tier 2: Pattern trap validation
        """
        scores = pd.read_parquet(f"{submission_path}/scores.parquet")

        trap_results = {}

        for trap in pattern_traps:
            trap_alert_ids = trap['alert_ids']
            expected_high_score = trap['expected_result']  # 1 = should score high

            trap_scores = scores[scores['alert_id'].isin(trap_alert_ids)]['score']

            if expected_high_score:
                # Trap alerts should score in top 10%
                threshold = scores['score'].quantile(0.9)
                trap_results[trap['trap_id']] = {
                    'passed': (trap_scores > threshold).mean() > 0.7,
                    'mean_score': float(trap_scores.mean())
                }
            else:
                # Trap alerts should score in bottom 50%
                threshold = scores['score'].median()
                trap_results[trap['trap_id']] = {
                    'passed': (trap_scores < threshold).mean() > 0.7,
                    'mean_score': float(trap_scores.mean())
                }

        trap_results['overall_passed'] = sum(
            r['passed'] for r in trap_results.values() if isinstance(r, dict)
        ) / len(pattern_traps) >= 0.8

        return trap_results

    def _compute_immediate_score(self, validation_results: dict) -> float:
        """
        Compute immediate score (0-1) from Tier 1 + Tier 2
        """
        tier1_score = sum(validation_results['tier1_integrity'].values()) / 6  # 6
checks

        trap_pass_rate = sum(
            r['passed'] for r in
validation_results['tier2_pattern_traps'].values()
            if isinstance(r, dict)
        ) / max(1, len(validation_results['tier2_pattern_traps']) - 1)  # -1 for
overall_passed

        # Weighted combination: 20% tier1, 30% tier2
        immediate_score = 0.2 * tier1_score + 0.3 * trap_pass_rate
```

```
        return float(immediate_score)

    def validate_at_t_plus_tau(
        self,
        miner_id: str,
        submission_path: str,
        ground_truth: pd.DataFrame
    ) -> dict:
        """
        Tier 3: Ground truth validation at T+τ
        """
        scores = pd.read_parquet(f"{submission_path}/scores.parquet")

        # Merge with ground truth
        eval_data = scores.merge(
            ground_truth[['alert_id', 'confirmed_illicit']],
            on='alert_id'
        )

        # Compute metrics
        from sklearn.metrics import roc_auc_score, brier_score_loss, log_loss,
ndcg_score

        y_true = eval_data['confirmed_illicit'].values
        y_pred = eval_data['score'].values

        metrics = {
            'auc': float(roc_auc_score(y_true, y_pred)),
            'brier': float(brier_score_loss(y_true, y_pred)),
            'logloss': float(log_loss(y_true, y_pred)),
            'ndcg_500': float(ndcg_score([y_true], [y_pred], k=500))
        }

        # Compute final score (50% weight for T+τ metrics)
        tau_score = (
            metrics['auc'] * 0.4 +
            (1 - metrics['brier']) * 0.3 +  # Invert brier (lower is better)
            metrics['ndcg_500'] * 0.3
        )

        return {
            'miner_id': miner_id,
            'metrics': metrics,
            'tau_score': float(tau_score)
        }

    def _compute_parquet_hash(self, path: str) -> str:
        """Compute hash of parquet file"""
        import hashlib
        sha256 = hashlib.sha256()
        with open(path, 'rb') as f:
            for chunk in iter(lambda: f.read(4096), b''):
                sha256.update(chunk)
```

```
        return sha256.hexdigest()

    def _verify_signature(self, receipt: dict) -> bool:
        """Verify cryptographic signature"""
        import nacl.signing
        import nacl.encoding

        # Reconstruct message
        message_data = {k: v for k, v in receipt.items() if k != 'signature'}
        message = json.dumps(message_data, sort_keys=True).encode()

        # Verify
        try:
            verify_key = nacl.signing.VerifyKey(
                receipt['miner_pubkey'],
                encoder=nacl.encoding.HexEncoder
            )
            verify_key.verify(message, bytes.fromhex(receipt['signature']))
            return True
        except:
            return False
```

---

## Complete Workflow Example

### Day 0: Validator Publishes Batch

```
# Validator creates daily batch
python -m validator.batch_publisher \
  --processing-date 2025-10-25 \
  --window-days 7 \
  --output /batches/2025-10-25/

# Files created:
# /batches/2025-10-25/
#   ├─ alerts.parquet (12,437 alerts)
#   ├─ features.parquet (234,567 addresses)
#   ├─ clusters.parquet (1,234 clusters)
#   ├─ money_flows.parquet (5,678,901 edges)
#   └─ META.json
```

### Day 0: Miner Downloads and Scores

```
# Miner downloads batch
wget https://validator.subnet.io/batches/2025-10-25.tar.gz
tar xzf 2025-10-25.tar.gz

# Run scoring in Docker
docker run --rm --network=none \
```

```
  -v $(pwd)/2025-10-25:/input:ro \
  -v $(pwd)/output:/output \
  miner-alert-scorer:v1

# Files created:
# output/
#   ├─ scores.parquet (12,437 rows with scores)
#   ├─ manifest.json
#   └─ receipt.json

# Upload submission
python -m miner.uploader \
  --submission-dir ./output \
  --batch-id 2025-10-25 \
  --validator-url https://validator.subnet.io
```

### Day 0: Validator Validates (T+0)

```
# Validator validates submission
validator = MinerSubmissionValidator()

results = validator.validate_submission(
    miner_id="miner_abc123",
    submission_path="/submissions/miner_abc123/2025-10-25",
    batch_meta=meta,
    pattern_traps=traps
)

# Results:
# {
#   'tier1_integrity': {'passed': True, ...},
#   'tier2_pattern_traps': {'overall_passed': True, ...},
#   'immediate_score': 0.47  # 20% tier1 + 30% tier2
# }
```

### Day 21: Validator Validates (T+τ)

```
# Load ground truth (outcomes from past 21 days)
ground_truth = build_ground_truth(
    batch_date='2025-10-25',
    tau_days=21
)

# Compute final metrics
final_results = validator.validate_at_t_plus_tau(
    miner_id="miner_abc123",
    submission_path="/submissions/miner_abc123/2025-10-25",
    ground_truth=ground_truth
)
```

```
# Results:
# {
#   'metrics': {
#     'auc': 0.87,
#     'brier': 0.12,
#     'logloss': 0.35,
#     'ndcg_500': 0.82
#   },
#   'tau_score': 0.85  # 50% weight
# }

# Total score = 0.47 (immediate) + 0.85 (tau) = 1.32
# Normalized: 1.32 / 2.0 = 0.66 (66th percentile)
```

---

## Summary

This implementation plan provides:

1. **Complete ML pipeline** for miners (training + inference)
2. **Deterministic execution** in Docker containers
3. **Multi-tier validation** framework (integrity, traps, ground truth)
4. **Production-ready code** with all three proposals implemented

**Miners need**:

- Historical data for training (provided by validator)
- Docker environment for reproducibility
- ML expertise for model improvement

**Validators need**:

- Daily batch generation pipeline
- Pattern trap system
- T+τ ground truth tracker
- Validation framework

**Timeline**: 8-12 weeks for miners to train initial models and start competing.