Open Source Framework & Score Aggregation

Miner Transparency and Score Utilization Strategies

Date: 2025-10-26
Purpose: Define open source requirements and score aggregation strategies

---

Overview

This document addresses:

1. **Open source framework** - Miners publish code, models, and datasets
2. **Score aggregation strategies** - How to use miner outputs
3. **Address/transaction risk scoring** - Mapping alert scores to entity risk

---

Part 1: Open Source Miner Framework

1.1 Miner Publication Requirements

Required Artifacts

```
┌─────────────────────────────────────────────────────────────┐
│                    MINER PUBLICATION                        │
├─────────────────────────────────────────────────────────────┤
│ 1. Source Code Repository (GitHub/GitLab)                   │
│    ├─ Training scripts                                      │
│    ├─ Inference code                                        │
│    ├─ Feature engineering                                   │
│    ├─ Model architecture                                    │
│    └─ README with full documentation                        │
│                                                             │
│ 2. Docker Image (Public Registry)                           │
│    ├─ Dockerfile                                            │
│    ├─ Reproducible build                                    │
│    ├─ Deterministic execution                               │
│    └─ Tagged by version                                     │
│                                                             │
│ 3. Model Weights                                            │
│    ├─ Trained model files (.txt, .pkl)                      │
│    ├─ Feature importance                                    │
│    ├─ Training metrics                                      │
│    └─ Validation results                                    │
│                                                             │
│ 4. Training Configuration                                   │
│    ├─ Hyperparameters                                       │
│    ├─ Seeds                                                 │
│    ├─ Training data specification                           │
```

```
│    └─ Feature selection                                    │
│                                                            │
│ 5. External Datasets (if any)                              │
│    ├─ Dataset sources                                      │
│    ├─ Preprocessing scripts                                │
│    ├─ License compliance                                   │
│    └─ Usage documentation                                  │
└────────────────────────────────────────────────────────────┘
```

File: `miner/MINER_MANIFEST.yaml`

```yaml
# Miner Publication Manifest
miner_id: "miner_abc123"
miner_name: "DeepAML Labs"
version: "1.2.3"

# Source Code
source_code:
  repository: "https://github.com/deepaml/alert-scorer"
  commit: "f7e1c1a2b3d4e5f6"
  license: "MIT"

# Docker Image
docker_image:
  registry: "ghcr.io"
  image: "deepaml/alert-scorer"
  tag: "v1.2.3"
  digest: "sha256:2a7c8d9e..."
  build_reproducible: true

# Model Weights
model:
  type: "LightGBM"
  version: "4.3.0"
  weights_file: "models/alert_scorer_v1.2.3.txt"
  weights_hash: "sha256:abc123..."
  training_date: "2025-10-15"

# Training Configuration
training:
  hyperparameters:
    num_leaves: 31
    learning_rate: 0.05
    num_iterations: 500
  features_used: 187  # Out of 200+ available
  feature_list: "config/features.json"
  seed: 42

# External Datasets
external_data:
  - name: "Chainalysis Sanctions List"
```

```
    source: "https://chainalysis.com/sanctions"
    license: "Commercial"
    usage: "Mixer proximity features"
    last_updated: "2025-10-01"

# Performance Metrics
performance:
  validation_auc: 0.87
  validation_brier: 0.12
  avg_inference_time_ms: 0.15  # per alert

# Contact
contact:
  team: "DeepAML Labs"
  email: "team@deepaml.io"
  website: "https://deepaml.io"
```

1.2 Repository Structure

```
miner-repo/
├── README.md                    # Full documentation
├── LICENSE                      # Open source license
├── MINER_MANIFEST.yaml          # Publication manifest
│
├── Dockerfile                   # Reproducible build
├── requirements.txt             # Python dependencies
├── docker-compose.yml           # Local testing
│
├── miner/                       # Source code
│   ├── __init__.py
│   ├── miner_main.py            # Main entry point
│   ├── models/
│   │   ├── alert_scorer.py
│   │   ├── alert_ranker.py
│   │   └── cluster_scorer.py
│   ├── features/
│   │   ├── feature_engineering.py
│   │   └── feature_selection.py
│   └── utils/
│       ├── data_loader.py
│       └── validators.py
│
├── training/                    # Training scripts
│   ├── train_alert_scorer.py
│   ├── train_ranker.py
│   └── hyperparameter_search.py
│
├── models/                      # Trained model weights
│   ├── alert_scorer_v1.2.3.txt
│   ├── alert_scorer_v1.2.3.txt.meta
│   └── feature_importance.json
```

```
│
├── config/                      # Configuration
│   ├── features.json           # Feature list
│   ├── hyperparameters.yaml    # Training config
│   └── external_data.yaml      # External dataset config
│
├── tests/                       # Unit tests
│   ├── test_scorer.py
│   ├── test_ranker.py
│   └── test_determinism.py
│
└── docs/                        # Documentation
    ├── architecture.md
    ├── training_guide.md
    └── reproducibility.md
```

1.3 Reproducibility Requirements

Determinism Checklist

```python
# File: miner/utils/determinism_enforcer.py

class DeterminismEnforcer:
    """
    Enforce deterministic execution
    """

    @staticmethod
    def enforce():
        """Set all random seeds"""
        import os
        import random
        import numpy as np
        import lightgbm as lgb

        # Python random
        random.seed(42)

        # NumPy
        np.random.seed(42)

        # Environment variables
        os.environ['PYTHONHASHSEED'] = '0'
        os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':4096:8'

        # LightGBM
        os.environ['LIGHTGBM_SEED'] = '42'

    @staticmethod
    def verify_reproducibility(func):
        """Decorator to verify function is deterministic"""
```

```
        def wrapper(*args, **kwargs):
            # Run twice
            result1 = func(*args, **kwargs)
            result2 = func(*args, **kwargs)

            # Compare
            import numpy as np
            if isinstance(result1, np.ndarray):
                assert np.array_equal(result1, result2), \
                    "Function is not deterministic!"

            return result1

        return wrapper
```

1.4 Validation Before Publication

```
# File: scripts/validate_publication.py

"""
Validate miner publication before submission
"""

import yaml
import json
import subprocess
from pathlib import Path
from loguru import logger

class PublicationValidator:
    """
    Validate miner meets publication requirements
    """

    def validate(self, repo_path: Path) -> dict:
        """Full validation"""
        checks = {}

        # Check 1: Manifest exists
        checks['manifest'] = self._check_manifest(repo_path)

        # Check 2: Docker builds successfully
        checks['docker_build'] = self._check_docker_build(repo_path)

        # Check 3: Determinism test
        checks['determinism'] = self._check_determinism(repo_path)

        # Check 4: License compliance
        checks['license'] = self._check_license(repo_path)

        # Check 5: Documentation
```

```
        checks['documentation'] = self._check_documentation(repo_path)

        # Check 6: Model weights exist
        checks['model_weights'] = self._check_model_weights(repo_path)

        all_passed = all(checks.values())

        return {
            'passed': all_passed,
            'checks': checks
        }

    def _check_determinism(self, repo_path: Path) -> bool:
        """
        Run same input twice, verify identical output
        """
        logger.info("Testing determinism...")

        # Run Docker container twice
        result1 = subprocess.run([
            'docker', 'run', '--rm',
            '-v', f'{repo_path}/test_data:/input:ro',
            '-v', f'{repo_path}/output1:/output',
            'miner:latest'
        ], capture_output=True)

        result2 = subprocess.run([
            'docker', 'run', '--rm',
            '-v', f'{repo_path}/test_data:/input:ro',
            '-v', f'{repo_path}/output2:/output',
            'miner:latest'
        ], capture_output=True)

        # Compare outputs
        import pandas as pd
        scores1 = pd.read_parquet(f'{repo_path}/output1/scores.parquet')
        scores2 = pd.read_parquet(f'{repo_path}/output2/scores.parquet')

        return scores1.equals(scores2)
```

Part 2: Score Aggregation Strategies

Strategy 1: Best Miner Wins (Champion Selection)

Concept

```
┌─────────────────────────────────────────────────────────────┐
│              STRATEGY 1: BEST MINER WINS                    │
├─────────────────────────────────────────────────────────────┤
│                                                             │
```

```
│  Daily Cycle:                                                    │
│  1. All miners submit scores                                     │
│  2. Validator evaluates all submissions (T+0 + T+τ)              │
│  3. Rank miners by performance                                   │
│  4. Select #1 miner as "champion"                                │
│  5. Use champion's scores as canonical                           │
│                                                                  │
│  Advantages:                                                     │
│  ✓ Simple to implement                                           │
│  ✓ Clear winner/loser                                            │
│  ✓ Fast (one model runs)                                         │
│  ✓ Easy to explain                                               │
│                                                                  │
│  Disadvantages:                                                  │
│  ✗ Winner-takes-all (discourages participation)                  │
│  ✗ No diversity benefit                                          │
│  ✗ Vulnerable if champion fails                                  │
│  ✗ Abrupt changes when champion changes                          │
└──────────────────────────────────────────────────────────────────┘
```

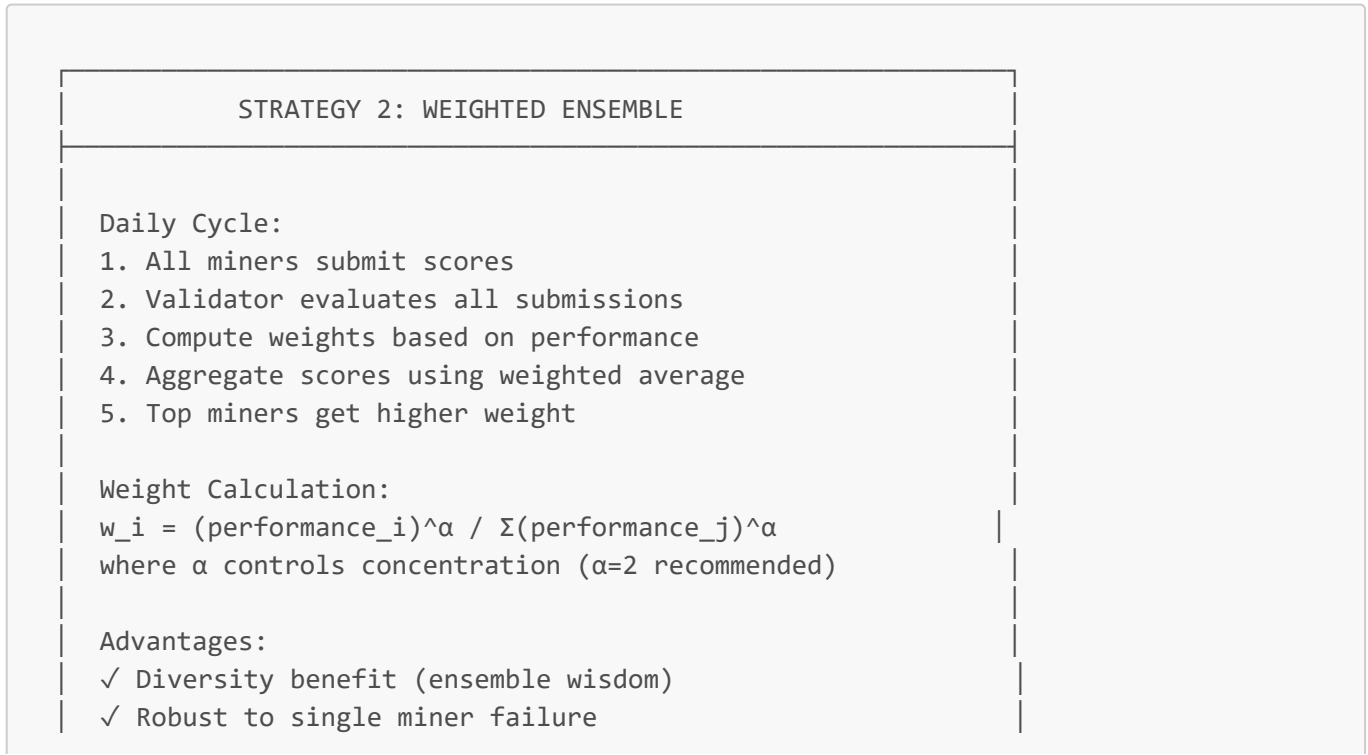
Implementation

```python
# File: validator/aggregation/champion_selector.py

class ChampionSelector:
    """
    Select best miner as canonical scorer
    """

    def select_champion(
        self,
        miner_scores: dict,  # {miner_id: performance_score}
        window_days: int = 30
    ) -> str:
        """
        Select champion miner based on rolling performance
        """
        # Get performance over last 30 days
        rolling_scores = {}

        for miner_id, history in miner_scores.items():
            recent = history[-window_days:]
            rolling_scores[miner_id] = {
                'mean': np.mean(recent),
                'std': np.std(recent),
                'min': np.min(recent),
                'consistency': 1.0 - (np.std(recent) / np.mean(recent))
            }

        # Composite score: 70% mean performance + 30% consistency
        composite_scores = {
```

```
            miner_id: 0.7 * metrics['mean'] + 0.3 * metrics['consistency']
            for miner_id, metrics in rolling_scores.items()
        }

        # Select best
        champion_id = max(composite_scores, key=composite_scores.get)

        logger.info(f"Champion selected: {champion_id} (score=
{composite_scores[champion_id]:.3f})")

        return champion_id

    def use_champion_scores(
        self,
        champion_id: str,
        submission_path: str
    ) -> pd.DataFrame:
        """
        Load and use champion's scores as canonical
        """
        scores = pd.read_parquet(f"
{submission_path}/{champion_id}/scores.parquet")

        # Add provenance
        scores['source_miner'] = champion_id
        scores['aggregation_method'] = 'champion'

        return scores
```

Strategy 2: Weighted Ensemble (Recommended)

Concept



```
│  ✓ Smooth weight changes                                      │
│  ✓ Rewards all top performers                                 │
│  ✓ Better generalization                                      │
│                                                               │
│  Disadvantages:                                               │
│  ✗ More complex                                               │
│  ✗ Slower (ensemble computation)                              │
│  ✗ Requires all submissions                                   │
└──────────────────────────────────────────────────────────────┘
```

Implementation

```python
# File: validator/aggregation/weighted_ensemble.py

class WeightedEnsembleAggregator:
    """
    Aggregate miner scores using weighted ensemble
    """

    def __init__(self, alpha: float = 2.0, min_weight: float = 0.01):
        self.alpha = alpha  # Concentration parameter
        self.min_weight = min_weight  # Minimum weight threshold

    def compute_weights(
        self,
        performance_scores: dict  # {miner_id: performance_score}
    ) -> dict:
        """
        Compute miner weights based on performance

        Uses power law: w_i = (score_i)^α / Σ(score_j)^α
        """
        # Apply power
        powered = {
            miner_id: score ** self.alpha
            for miner_id, score in performance_scores.items()
            if score > 0
        }

        # Normalize
        total = sum(powered.values())
        weights = {
            miner_id: value / total
            for miner_id, value in powered.items()
        }

        # Filter low weights
        weights = {
            miner_id: weight
            for miner_id, weight in weights.items()
            if weight >= self.min_weight
```

```
        }

        # Renormalize after filtering
        total = sum(weights.values())
        weights = {
            miner_id: weight / total
            for miner_id, weight in weights.items()
        }

        logger.info(f"Computed weights for {len(weights)} miners")
        for miner_id, weight in sorted(weights.items(), key=lambda x: x[1],
reverse=True):
            logger.info(f"  {miner_id}: {weight:.3f}")

        return weights

    def aggregate_scores(
        self,
        miner_submissions: dict,  # {miner_id: scores_df}
        weights: dict  # {miner_id: weight}
    ) -> pd.DataFrame:
        """
        Compute weighted average of miner scores
        """
        # Get common alert_ids (all miners must score these)
        alert_ids = set(miner_submissions[list(miner_submissions.keys())[0]]
['alert_id'])
        for scores_df in miner_submissions.values():
            alert_ids &= set(scores_df['alert_id'])

        logger.info(f"Computing ensemble for {len(alert_ids)} common alerts")

        # Aggregate
        ensemble_scores = []

        for alert_id in alert_ids:
            # Get scores from all miners
            miner_scores = {}
            for miner_id, scores_df in miner_submissions.items():
                if miner_id in weights:
                    score = scores_df[scores_df['alert_id'] == alert_id]
['score'].iloc[0]
                    miner_scores[miner_id] = score

            # Weighted average
            ensemble_score = sum(
                weights[miner_id] * score
                for miner_id, score in miner_scores.items()
            )

            # Metadata
            ensemble_scores.append({
                'alert_id': alert_id,
                'score': ensemble_score,
```

```
                'num_miners': len(miner_scores),
                'score_std': np.std(list(miner_scores.values())),
                'contributing_miners': list(miner_scores.keys())
            })

        return pd.DataFrame(ensemble_scores)

    def save_ensemble(
        self,
        ensemble_scores: pd.DataFrame,
        weights: dict,
        output_path: str
    ):
        """
        Save ensemble results with metadata
        """
        # Save scores
        ensemble_scores.to_parquet(f"{output_path}/ensemble_scores.parquet",
index=False)

        # Save weights
        with open(f"{output_path}/ensemble_weights.json", 'w') as f:
            json.dump(weights, f, indent=2)

        # Save metadata
        metadata = {
            'timestamp': pd.Timestamp.now().isoformat(),
            'num_alerts': len(ensemble_scores),
            'num_miners': len(weights),
            'alpha': self.alpha,
            'top_5_miners': sorted(weights.items(), key=lambda x: x[1],
reverse=True)[:5]
        }

        with open(f"{output_path}/ensemble_metadata.json", 'w') as f:
            json.dump(metadata, f, indent=2)

        logger.info(f"Ensemble saved to {output_path}")
```

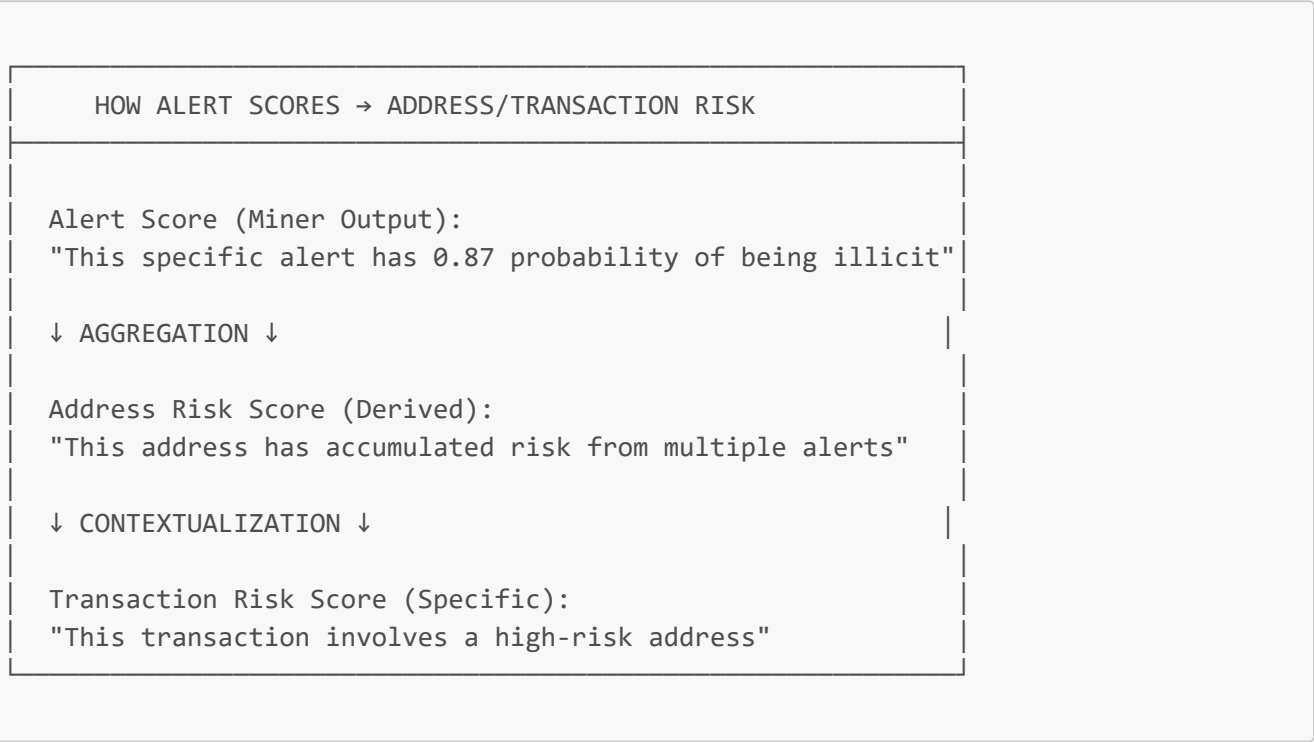
Strategy Comparison

Aspect	Strategy 1: Champion	Strategy 2: Ensemble
Performance	Good (best single model)	Better (ensemble wisdom)
Robustness	Vulnerable to champion failure	Robust to individual failures
Simplicity	Very simple	Moderate complexity
Speed	Fast (one model)	Slower (aggregate)
Miner Incentives	Winner-takes-all	Rewards top performers

Aspect	Strategy 1: Champion	Strategy 2: Ensemble
Generalization	Depends on champion	Better (diverse models)
Stability	Can have abrupt changes	Smooth transitions
Recommended For	Prototype/testing	Production

Part 3: Mapping Alert Scores to Address/Transaction Risk

3.1 The Relationship



3.2 Address Risk Scoring

File: `validator/scoring/address_risk_scorer.py`

```
"""
Compute address-level risk scores from alert scores
"""

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

class AddressRiskScorer:
    """
    Aggregate alert scores to address-level risk
    """

    def compute_address_risk(
        self,
```

```
        alert_scores: pd.DataFrame,  # From ensemble or champion
        alerts: pd.DataFrame,  # Original alerts
        window_days: int = 7
    ) -> pd.DataFrame:
        """
        Compute risk score for each address

        Strategy: Combine multiple alerts per address
        """
        # Merge scores with alert metadata
        scored_alerts = alerts.merge(
            alert_scores[['alert_id', 'score']],
            on='alert_id'
        )

        # Group by address
        address_risks = []

        for address in scored_alerts['address'].unique():
            addr_alerts = scored_alerts[scored_alerts['address'] == address]

            # Compute address risk
            risk_score = self._aggregate_alert_scores(addr_alerts)

            address_risks.append({
                'address': address,
                'risk_score': risk_score,
                'num_alerts': len(addr_alerts),
                'max_alert_score': addr_alerts['score'].max(),
                'mean_alert_score': addr_alerts['score'].mean(),
                'alert_types': addr_alerts['typology_type'].tolist(),
                'window_days': window_days
            })

        return pd.DataFrame(address_risks)

    def _aggregate_alert_scores(self, addr_alerts: pd.DataFrame) -> float:
        """
        Aggregate multiple alert scores for an address

        Methods:
        1. Maximum (most conservative)
        2. Average (balanced)
        3. Weighted average by severity
        4. Probabilistic independence
        """
        scores = addr_alerts['score'].values
        severities = addr_alerts['severity'].map({
            'low': 0.25, 'medium': 0.5, 'high': 0.75, 'critical': 1.0
        }).values

        # Method 1: Maximum (if ANY alert is high risk, address is high risk)
        max_score = np.max(scores)
```

```
        # Method 2: Average
        avg_score = np.mean(scores)

        # Method 3: Weighted by severity
        weighted_score = np.average(scores, weights=severities)

        # Method 4: Probabilistic independence
        # P(at least one alert is true) = 1 - Π(1 - p_i)
        prob_score = 1.0 - np.prod(1.0 - scores)

        # Final score: weighted combination
        # 40% max + 30% weighted + 30% probabilistic
        final_score = (
            0.4 * max_score +
            0.3 * weighted_score +
            0.3 * prob_score
        )

        return float(np.clip(final_score, 0, 1))
```

3.3 Transaction Risk Scoring

File: `validator/scoring/transaction_risk_scorer.py`

```
"""
Compute transaction-level risk scores
"""

class TransactionRiskScorer:
    """
    Score individual transactions based on address risks
    """

    def score_transaction(
        self,
        transaction_id: str,
        from_address: str,
        to_address: str,
        amount_usd: float,
        address_risks: pd.DataFrame,
        alert_scores: pd.DataFrame
    ) -> dict:
        """
        Compute risk score for a specific transaction
        """
        # Get address risks
        from_risk = self._get_address_risk(from_address, address_risks)
        to_risk = self._get_address_risk(to_address, address_risks)

        # Get alerts related to this transaction (if any)
        tx_alerts = self._get_transaction_alerts(
```

```
            transaction_id,
            alert_scores
        )

        # Compute base risk
        # Use maximum of from/to address risk
        base_risk = max(from_risk, to_risk)

        # Adjust for transaction-specific factors
        amount_factor = self._compute_amount_factor(amount_usd)
        alert_factor = self._compute_alert_factor(tx_alerts)

        # Final transaction risk
        tx_risk = base_risk * amount_factor * alert_factor
        tx_risk = np.clip(tx_risk, 0, 1)

        return {
            'transaction_id': transaction_id,
            'risk_score': float(tx_risk),
            'from_address': from_address,
            'to_address': to_address,
            'from_address_risk': from_risk,
            'to_address_risk': to_risk,
            'amount_usd': amount_usd,
            'amount_factor': amount_factor,
            'alert_factor': alert_factor,
            'related_alerts': len(tx_alerts),
            'risk_category': self._categorize_risk(tx_risk)
        }

    def _get_address_risk(
        self,
        address: str,
        address_risks: pd.DataFrame
    ) -> float:
        """Get risk score for address"""
        addr_data = address_risks[address_risks['address'] == address]

        if len(addr_data) == 0:
            return 0.1  # Default low risk for unknown addresses

        return float(addr_data['risk_score'].iloc[0])

    def _compute_amount_factor(self, amount_usd: float) -> float:
        """
        Adjust risk based on transaction amount
        Higher amounts = higher concern
        """
        if amount_usd < 1000:
            return 0.8
        elif amount_usd < 10000:
            return 1.0
        elif amount_usd < 100000:
            return 1.2
```

```
        else:
            return 1.5

    def _compute_alert_factor(self, tx_alerts: list) -> float:
        """
        Adjust risk based on transaction-specific alerts
        """
        if len(tx_alerts) == 0:
            return 1.0

        # If transaction has alerts, boost risk
        max_alert_score = max(alert['score'] for alert in tx_alerts)

        # 1.0 to 2.0 multiplier based on max alert
        return 1.0 + max_alert_score

    def _categorize_risk(self, risk_score: float) -> str:
        """Categorize risk into bands"""
        if risk_score < 0.3:
            return 'low'
        elif risk_score < 0.6:
            return 'medium'
        elif risk_score < 0.8:
            return 'high'
        else:
            return 'critical'
```

3.4 Complete Flow Example

```
# Daily Risk Scoring Pipeline

from validator.aggregation.weighted_ensemble import WeightedEnsembleAggregator
from validator.scoring.address_risk_scorer import AddressRiskScorer
from validator.scoring.transaction_risk_scorer import TransactionRiskScorer

def daily_risk_scoring_pipeline(processing_date: str):
    """
    Complete pipeline from miner submissions to transaction scores
    """

    # Step 1: Load miner submissions
    miner_submissions = load_all_miner_submissions(processing_date)
    miner_weights = load_miner_weights(processing_date)

    # Step 2: Aggregate alert scores (Strategy 2: Ensemble)
    aggregator = WeightedEnsembleAggregator(alpha=2.0)
    alert_scores = aggregator.aggregate_scores(
        miner_submissions,
        miner_weights
    )
```

```
    # Step 3: Compute address-level risk
    address_scorer = AddressRiskScorer()
    address_risks = address_scorer.compute_address_risk(
        alert_scores,
        alerts=load_alerts(processing_date),
        window_days=7
    )

    # Step 4: Score transactions on demand
    tx_scorer = TransactionRiskScorer()

    # Example: Score a specific transaction
    tx_score = tx_scorer.score_transaction(
        transaction_id="0xabc123...",
        from_address="0x123...",
        to_address="0x456...",
        amount_usd=50000,
        address_risks=address_risks,
        alert_scores=alert_scores
    )

    # Save results
    save_address_risks(address_risks, processing_date)

    return {
        'alert_scores': alert_scores,
        'address_risks': address_risks,
        'example_tx_score': tx_score
    }
```

3.5 API Endpoints for Risk Queries

```
# File: validator/api/risk_api.py

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class AddressRiskRequest(BaseModel):
    address: str
    window_days: int = 7

class TransactionRiskRequest(BaseModel):
    from_address: str
    to_address: str
    amount_usd: float

@app.get("/api/v1/risk/address/{address}")
def get_address_risk(address: str, window_days: int = 7):
    """
```

```
    Get risk score for an address
    """
    address_risks = load_latest_address_risks(window_days)

    addr_data = address_risks[address_risks['address'] == address]

    if len(addr_data) == 0:
        return {
            'address': address,
            'risk_score': 0.1,
            'risk_category': 'unknown',
            'message': 'No alerts found for this address'
        }

    return addr_data.iloc[0].to_dict()

@app.post("/api/v1/risk/transaction")
def score_transaction(request: TransactionRiskRequest):
    """
    Score a transaction in real-time
    """
    address_risks = load_latest_address_risks()
    alert_scores = load_latest_alert_scores()

    scorer = TransactionRiskScorer()

    tx_score = scorer.score_transaction(
        transaction_id=None,  # Real-time, no ID yet
        from_address=request.from_address,
        to_address=request.to_address,
        amount_usd=request.amount_usd,
        address_risks=address_risks,
        alert_scores=alert_scores
    )

    return tx_score
```

Summary

Open Source Framework

- **All miners publish**: code, Docker images, models, parameters, datasets
- **Full transparency**: reproducible builds, determinism verification
- **Publication checklist**: manifest, tests, documentation

Score Aggregation

- **Strategy 1 (Champion)**: Use best miner's scores - simple but vulnerable
- **Strategy 2 (Ensemble)**: Weighted average - robust, better performance ☑ **RECOMMENDED**

Risk Score Mapping

```
Alert Scores (Miner Output)
  ↓
Address Risk Scores (Aggregate multiple alerts per address)
  ↓
Transaction Risk Scores (Combine address risks + transaction context)
```

Key Insight: Alert scores are the foundation. Address and transaction scores are derived by aggregating and contextualizing alert-level predictions.