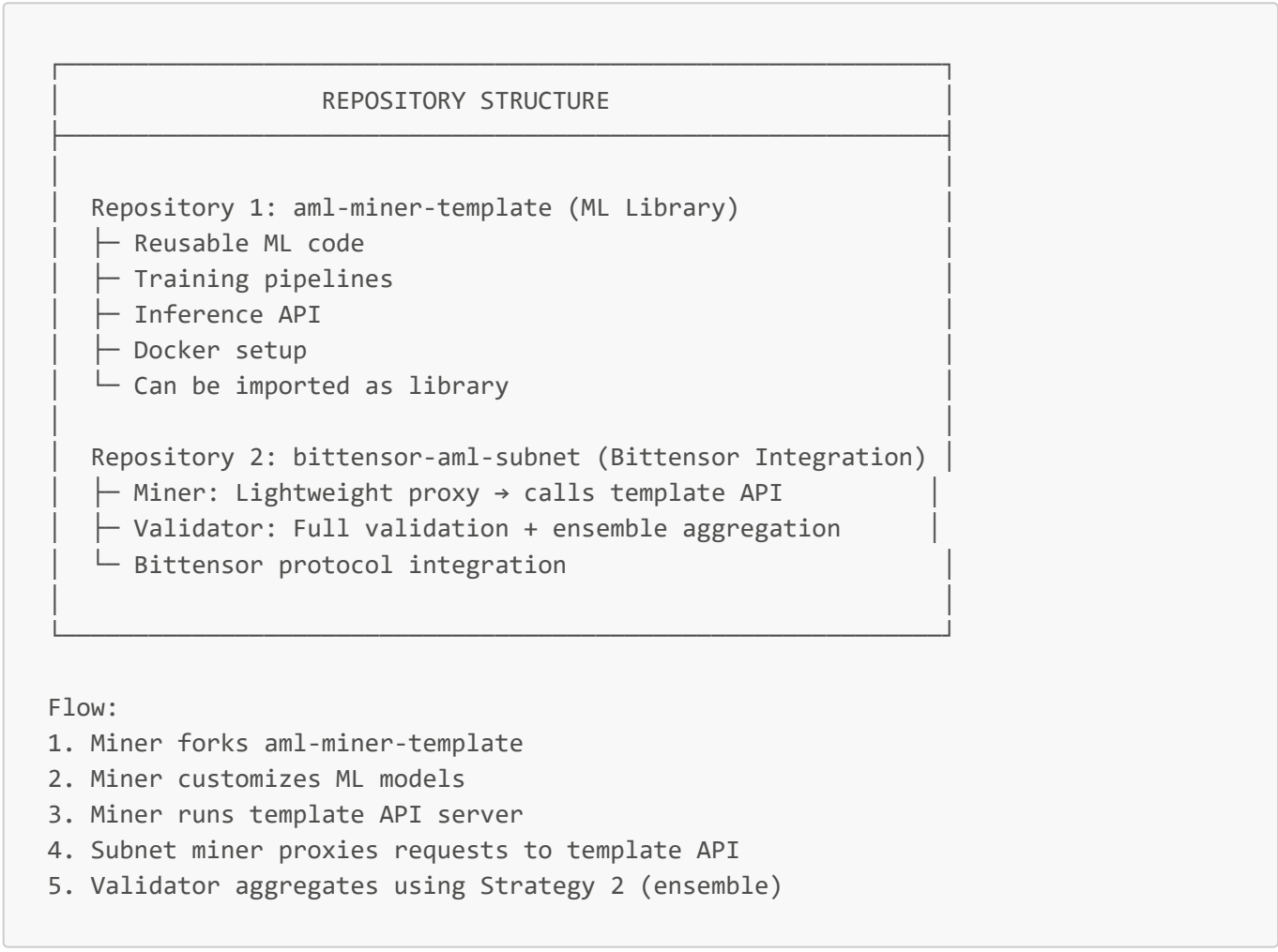
Miner Template Repository & Subnet Proxy Architecture

Separation of ML Library from Bittensor Integration

Date: 2025-10-26
Purpose: Define template repository structure and subnet proxy pattern
Strategy: Strategy 2 (Weighted Ensemble) confirmed ☑

Architecture Overview



Repository 1: aml-miner-template

Purpose

Reusable ML library that provides all the heavy lifting for alert scoring.

Repository Structure

```
aml-miner-template/
├── README.md                    # Quick start guide
├── LICENSE                      # MIT
├── pyproject.toml              # Python package config
├── setup.py                     # Package installation
├── .env.example                 # Configuration template
│
├── Dockerfile                   # Production image
├── docker-compose.yml          # Local development
├── requirements.txt             # Dependencies
│
├── aml_miner/                   # Main package (importable)
│   ├── __init__.py
│   ├── version.py              # Version info
│   │
│   ├── api/                    # FastAPI server
│   │   ├── __init__.py
│   │   ├── server.py           # Main API server
│   │   ├── routes.py           # API endpoints
│   │   └── schemas.py          # Pydantic models
│   │
│   ├── models/                 # ML models
│   │   ├── __init__.py
│   │   ├── alert_scorer.py
│   │   ├── alert_ranker.py
│   │   ├── cluster_scorer.py
│   │   └── base_model.py       # Abstract base class
│   │
│   ├── training/               # Training pipelines
│   │   ├── __init__.py
│   │   ├── train_scorer.py
│   │   ├── train_ranker.py
│   │   └── hyperparameter_tuner.py
│   │
│   ├── features/               # Feature engineering
│   │   ├── __init__.py
│   │   ├── feature_builder.py
│   │   └── feature_selector.py
│   │
│   ├── utils/                  # Utilities
│   │   ├── __init__.py
│   │   ├── data_loader.py
│   │   ├── validators.py
│   │   └── determinism.py
│   │
│   └── config/                 # Configuration
│       ├── __init__.py
│       ├── settings.py         # Pydantic settings
│       └── model_config.yaml   # Model hyperparameters
│
├── trained_models/              # Pretrained models
│   ├── alert_scorer_v1.0.0.txt
│   ├── alert_ranker_v1.0.0.txt
```

```
│   └── model_metadata.json
│
├── scripts/                      # Utility scripts
│   ├── download_batch.sh        # Download SOT batch
│   ├── train_models.py          # Train all models
│   └── validate_submission.py   # Test before submit
│
├── tests/                        # Unit tests
│   ├── test_models.py
│   ├── test_api.py
│   └── test_determinism.py
│
└── docs/                         # Documentation
    ├── quickstart.md
    ├── training_guide.md
    ├── customization.md
    └── api_reference.md
```

Key Files

aml_miner/api/server.py

```python
"""
FastAPI server for miner inference
This is the main API that subnet miner will call
"""

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import pandas as pd
from typing import List, Dict
from loguru import logger

from aml_miner.models.alert_scorer import AlertScorerModel
from aml_miner.models.alert_ranker import AlertRankerModel
from aml_miner.models.cluster_scorer import ClusterScorerModel
from aml_miner.config.settings import Settings

# Load settings
settings = Settings()

# Initialize FastAPI
app = FastAPI(
    title="AML Miner API",
    description="Alert scoring, ranking, and cluster assessment",
    version="1.0.0"
)

# Load models at startup
alert_scorer = None
alert_ranker = None
```

```
cluster_scorer = None

@app.on_event("startup")
async def startup_event():
    """Load trained models"""
    global alert_scorer, alert_ranker, cluster_scorer

    logger.info("Loading models...")

    alert_scorer = AlertScorerModel()
    alert_scorer.load_model(settings.ALERT_SCORER_PATH)

    alert_ranker = AlertRankerModel()
    alert_ranker.load_model(settings.ALERT_RANKER_PATH)

    cluster_scorer = ClusterScorerModel()
    cluster_scorer.load_model(settings.CLUSTER_SCORER_PATH)

    logger.info("✓ Models loaded")

# Request/Response schemas
class BatchData(BaseModel):
    alerts: List[Dict]
    features: List[Dict]
    clusters: List[Dict]
    money_flows: List[Dict]

class ScoreResponse(BaseModel):
    alert_id: str
    score: float
    model_version: str
    latency_ms: int
    explain_json: str

@app.get("/health")
def health_check():
    """Health check endpoint"""
    return {
        "status": "healthy",
        "models_loaded": {
            "alert_scorer": alert_scorer is not None,
            "alert_ranker": alert_ranker is not None,
            "cluster_scorer": cluster_scorer is not None
        }
    }

@app.get("/version")
def get_version():
    """Get API and model versions"""
    return {
        "api_version": "1.0.0",
        "alert_scorer_version": alert_scorer.model_version if alert_scorer else
None,
        "alert_ranker_version": alert_ranker.model_version if alert_ranker else
```

```
    None,
        "cluster_scorer_version": cluster_scorer.model_version if cluster_scorer
else None
    }

@app.post("/score/alerts", response_model=List[ScoreResponse])
def score_alerts(batch: BatchData):
    """
    Score all alerts in batch
    This is the main endpoint called by subnet miner
    """
    import time
    t0 = time.time()

    # Convert to DataFrames
    alerts_df = pd.DataFrame(batch.alerts)
    features_df = pd.DataFrame(batch.features)
    clusters_df = pd.DataFrame(batch.clusters)

    # Prepare features
    X = alert_scorer.prepare_features(alerts_df, features_df, clusters_df)

    # Score
    scores = alert_scorer.predict(X)

    # Create explanations
    explanations = alert_scorer.create_explanations(X, scores)

    # Compute latency
    elapsed_ms = int((time.time() - t0) * 1000)
    latency_per_alert = elapsed_ms // max(1, len(alerts_df))

    # Format response
    results = []
    for i, alert_id in enumerate(alerts_df['alert_id']):
        results.append(ScoreResponse(
            alert_id=alert_id,
            score=float(scores[i]),
            model_version=alert_scorer.model_version,
            latency_ms=latency_per_alert,
            explain_json=explanations[i]
        ))

    logger.info(f"Scored {len(results)} alerts in {elapsed_ms}ms")

    return results

@app.post("/rank/alerts")
def rank_alerts(batch: BatchData):
    """
    Rank alerts by priority
    """
    # Similar implementation to score_alerts
    # but returns ranked list
```

```
    pass

@app.post("/score/clusters")
def score_clusters(batch: BatchData):
    """
    Score alert clusters
    """
    # Cluster scoring implementation
    pass

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(
        app,
        host="0.0.0.0",
        port=8000,
        log_level="info"
    )
```

pyproject.toml

```
[build-system]
requires = ["setuptools>=45", "wheel", "setuptools_scm[toml]>=6.2"]
build-backend = "setuptools.build_meta"

[project]
name = "aml-miner-template"
version = "1.0.0"
description = "Template for building AML alert scoring miners"
readme = "README.md"
requires-python = ">=3.11"
license = {text = "MIT"}
authors = [
    {name = "AML Subnet Team", email = "team@aml-subnet.io"}
]

dependencies = [
    "fastapi>=0.104.0",
    "uvicorn[standard]>=0.24.0",
    "pydantic>=2.5.0",
    "pydantic-settings>=2.1.0",
    "pandas>=2.1.0",
    "numpy>=1.26.0",
    "scikit-learn>=1.3.0",
    "lightgbm>=4.3.0",
    "shap>=0.44.0",
    "loguru>=0.7.2",
    "pynacl>=1.5.0",
    "pyarrow>=14.0.0",
    "httpx>=0.25.0"
]
```

```
[project.optional-dependencies]
dev = [
    "pytest>=7.4.0",
    "pytest-cov>=4.1.0",
    "black>=23.11.0",
    "ruff>=0.1.6",
    "mypy>=1.7.0"
]

[project.scripts]
aml-miner-serve = "aml_miner.api.server:main"
aml-miner-train = "aml_miner.training.train_scorer:main"
```

README.md

```
# AML Miner Template

Reusable template for building alert scoring miners for the AML subnet.

## Quick Start

### 1. Fork and Clone
```bash
git clone https://github.com/your-username/aml-miner-template
cd aml-miner-template
```

2. Install

```
pip install -e .
```

3. Download Pretrained Models (Optional)

```
# Use provided models
ls trained_models/
# alert_scorer_v1.0.0.txt
# alert_ranker_v1.0.0.txt
```

4. Run API Server

```
aml-miner-serve
# Server starts at http://localhost:8000
```
```

5. Test

```
curl http://localhost:8000/health
# {"status": "healthy", "models_loaded": {...}}
```

Customization

Train Your Own Models

```
# Download training data
scripts/download_batch.sh --start-date 2025-01-01 --end-date 2025-10-01

# Train
python aml_miner/training/train_scorer.py \
    --data-dir ./data \
    --output trained_models/my_scorer_v1.txt
```

Customize Features

Edit `aml_miner/features/feature_builder.py` to add your own features.

Hyperparameter Tuning

```
python aml_miner/training/hyperparameter_tuner.py \
    --trials 100 \
    --output best_params.yaml
```

API Reference

POST /score/alerts

Score all alerts in batch.

Request:

```
{
  "alerts": [...],
  "features": [...],
  "clusters": [...],
  "money_flows": [...]
}
```

Response:

```
[
  {
    "alert_id": "alert_123",
    "score": 0.87,
    "model_version": "v1.0.0",
    "latency_ms": 0.15,
    "explain_json": "{...}"
  }
]
```

License

MIT

```
---

## Repository 2: bittensor-aml-subnet

### Purpose
**Bittensor integration layer** - lightweight proxies and validation logic.

### Repository Structure
```

bittensor-aml-subnet/ ├── README.md ├── LICENSE ├── requirements.txt │ ├── neurons/ # Bittensor neurons │ ├── **init**.py │ ├── miner.py # Miner neuron (PROXY) │ └── validator.py # Validator neuron (FULL LOGIC) │ ├── protocol/ # Bittensor protocol │ ├── **init**.py │ ├── synapse.py # Synapse definitions │ └── validator_utils.py │ ├── validation/ # Validator-only code │ ├── **init**.py │ ├── ensemble_aggregator.py # Strategy 2: Weighted Ensemble │ ├── tier1_validator.py # Integrity checks │ ├── tier2_validator.py # Pattern traps │ └── tier3_validator.py # T+τ ground truth │ ├── config/ # Configuration │ ├── **init**.py │ └── subnet_config.yaml │ └── scripts/ # Deployment scripts ├── run_miner.sh └── run_validator.sh

```
### Miner Neuron (Proxy Pattern)

#### `neurons/miner.py`

```python
"""
Lightweight miner neuron - proxies to aml-miner-template API
"""

import bittensor as bt
from typing import List
import httpx
from loguru import logger
```
```

```
class AMLMiner(bt.BaseNeuron):
    """
    Miner neuron that proxies to aml-miner-template API
    """

    def __init__(self, config=None):
        super().__init__(config)

        # Miner template API endpoint (local or remote)
        self.miner_api_url = config.miner.api_url  # e.g., http://localhost:8000

        # HTTP client
        self.client = httpx.AsyncClient(timeout=120.0)

        logger.info(f"Miner initialized, API: {self.miner_api_url}")

    async def forward(self, synapse: bt.Synapse) -> bt.Synapse:
        """
        Forward request to miner template API
        """
        try:
            # Extract batch data from synapse
            batch_data = {
                "alerts": synapse.alerts,
                "features": synapse.features,
                "clusters": synapse.clusters,
                "money_flows": synapse.money_flows
            }

            # Call miner template API
            response = await self.client.post(
                f"{self.miner_api_url}/score/alerts",
                json=batch_data
            )

            response.raise_for_status()

            # Parse response
            scores = response.json()

            # Populate synapse with scores
            synapse.scores = scores
            synapse.miner_id = self.config.wallet.hotkey.ss58_address
            synapse.model_version = scores[0]['model_version'] if scores else None

            logger.info(f"Scored {len(scores)} alerts")

        except Exception as e:
            logger.error(f"Error calling miner API: {e}")
            synapse.scores = []

        return synapse
```

```
    async def blacklist(self, synapse: bt.Synapse) -> bool:
        """Blacklist logic"""
        # Accept all valid requests
        return False

    async def priority(self, synapse: bt.Synapse) -> float:
        """Priority logic"""
        return 1.0

def main():
    """Run miner"""
    config = bt.config()
    config.miner.api_url = "http://localhost:8000"  # Template API URL

    miner = AMLMiner(config)

    logger.info("Starting miner neuron...")
    bt.logging.info("Miner started. Waiting for requests...")

    # Run axon
    axon = bt.axon(wallet=miner.wallet, config=miner.config)
    axon.attach(forward_fn=miner.forward)
    axon.serve(netuid=miner.config.netuid)
    axon.start()

    # Keep alive
    import time
    while True:
        time.sleep(60)

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

Validator Neuron (Full Logic)

neurons/validator.py

```
"""
Validator neuron with ensemble aggregation (Strategy 2)
"""

import bittensor as bt
from typing import List, Dict
import pandas as pd
from loguru import logger

from validation.ensemble_aggregator import WeightedEnsembleAggregator
from validation.tier1_validator import Tier1Validator
from validation.tier2_validator import Tier2Validator
from validation.tier3_validator import Tier3Validator
```

```python
class AMLValidator(bt.BaseValidator):
    """
    Validator with ensemble aggregation
    """

    def __init__(self, config=None):
        super().__init__(config)

        # Aggregation strategy (Strategy 2: Weighted Ensemble)
        self.aggregator = WeightedEnsembleAggregator(alpha=2.0)

        # Validators
        self.tier1 = Tier1Validator()
        self.tier2 = Tier2Validator()
        self.tier3 = Tier3Validator()

        # Miner performance tracking
        self.miner_scores = {}  # {miner_id: performance_history}

        logger.info("Validator initialized with Strategy 2 (Weighted Ensemble)")

    async def forward(self):
        """
        Query all miners and aggregate results
        """
        # 1. Get active miners
        miners = self.metagraph.axons
        logger.info(f"Querying {len(miners)} miners...")

        # 2. Create synapse with batch data
        synapse = self.create_synapse()

        # 3. Query all miners in parallel
        responses = await self.dendrite.query(
            miners,
            synapse=synapse,
            timeout=120.0
        )

        # 4. Validate submissions (Tier 1 + Tier 2)
        validated_submissions = {}
        immediate_scores = {}

        for miner_axon, response in zip(miners, responses):
            miner_id = miner_axon.hotkey

            # Tier 1: Integrity
            tier1_result = self.tier1.validate(response)

            if not tier1_result['passed']:
                logger.warning(f"Miner {miner_id} failed Tier 1")
                continue

            # Tier 2: Pattern traps
```

```
            tier2_result = self.tier2.validate(response, self.pattern_traps)

            # Compute immediate score (0-0.5 range, out of 1.0 total)
            immediate_score = (
                0.2 * tier1_result['score'] +
                0.3 * tier2_result['score']
            )

            validated_submissions[miner_id] = response.scores
            immediate_scores[miner_id] = immediate_score

            logger.info(f"Miner {miner_id}: immediate_score=
{immediate_score:.3f}")

        # 5. Aggregate using Strategy 2 (Weighted Ensemble)
        if len(validated_submissions) > 0:
            # Compute weights based on historical performance
            weights =
self.aggregator.compute_weights(self.get_miner_performance())

            # Aggregate scores
            ensemble_scores = self.aggregator.aggregate_scores(
                validated_submissions,
                weights
            )

            # Save ensemble as canonical
            self.save_canonical_scores(ensemble_scores)

            logger.info(f"✓ Ensemble aggregated from {len(validated_submissions)}
miners")

        # 6. Update miner scores
        for miner_id, score in immediate_scores.items():
            if miner_id not in self.miner_scores:
                self.miner_scores[miner_id] = []
            self.miner_scores[miner_id].append(score)

        # 7. Set weights (temporary, will be updated at T+τ)
        self.set_miner_weights(immediate_scores)

    def get_miner_performance(self) -> Dict[str, float]:
        """
        Get rolling performance scores for all miners
        """
        performance = {}

        for miner_id, history in self.miner_scores.items():
            # Use last 30 days
            recent = history[-30:] if len(history) > 30 else history

            if recent:
                performance[miner_id] = sum(recent) / len(recent)
```

```
        return performance

    def set_miner_weights(self, scores: Dict[str, float]):
        """
        Set miner weights on chain
        """
        # Convert scores to weights
        # This is temporary, will be updated with Tier 3 scores

        miner_uids = []
        weights = []

        for miner_id, score in scores.items():
            uid = self.get_miner_uid(miner_id)
            if uid is not None:
                miner_uids.append(uid)
                weights.append(score)

        # Set weights
        if miner_uids:
            self.subtensor.set_weights(
                wallet=self.wallet,
                netuid=self.config.netuid,
                uids=miner_uids,
                weights=weights
            )

            logger.info(f"Set weights for {len(miner_uids)} miners")

    def create_synapse(self):
        """
        Create synapse with batch data
        """
        # Load today's batch
        batch = self.load_daily_batch()

        synapse = bt.Synapse(
            alerts=batch['alerts'],
            features=batch['features'],
            clusters=batch['clusters'],
            money_flows=batch['money_flows']
        )

        return synapse

def main():
    """Run validator"""
    config = bt.config()

    validator = AMLValidator(config)

    logger.info("Starting validator neuron...")

    # Run validation loop
```

```
    import asyncio
    while True:
        asyncio.run(validator.forward())

        # Wait before next cycle (e.g., hourly)
        import time
        time.sleep(3600)

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

Deployment Flow

For Miners

```
# Step 1: Setup miner template (one-time)
git clone https://github.com/aml-subnet/aml-miner-template
cd aml-miner-template
pip install -e .

# Step 2: Train your models (optional)
python aml_miner/training/train_scorer.py \
    --data-dir ./data \
    --output trained_models/my_scorer_v1.txt

# Step 3: Start miner template API
aml-miner-serve
# Runs on http://localhost:8000

# Step 4: Start subnet miner (in separate terminal)
cd ../bittensor-aml-subnet
python neurons/miner.py \
    --wallet.name my_wallet \
    --wallet.hotkey my_hotkey \
    --miner.api_url http://localhost:8000 \
    --netuid 42
```

For Validators

```
# Validators only need subnet repository
git clone https://github.com/aml-subnet/bittensor-aml-subnet
cd bittensor-aml-subnet

pip install -r requirements.txt

# Run validator
python neurons/validator.py \
    --wallet.name my_validator \
```

```
    --wallet.hotkey my_hotkey \
    --netuid 42
```

Benefits of This Architecture

For Miners

- ☑ **Easy to customize** - Fork template, modify ML models
- ☑ **Reusable library** - Can import `aml_miner` in any project
- ☑ **Separation of concerns** - ML code separate from Bittensor
- ☑ **Easy testing** - Test API independently
- ☑ **Version control** - Template updates don't break subnet

For Subnet

- ☑ **Lightweight miner** - Just a proxy (50 lines of code)
- ☑ **Full validator logic** - Ensemble, validation, scoring
- ☑ **Easy deployment** - No ML dependencies in subnet repo
- ☑ **Clean separation** - Bittensor protocol vs ML library

For Ecosystem

- ☑ **Innovation** - Miners can publish template improvements
- ☑ **Competition** - Easy to compare ML approaches
- ☑ **Transparency** - All code is open source
- ☑ **Upgradability** - Template can evolve independently

Summary

Two Repositories

1. **aml-miner-template**: Reusable ML library with FastAPI server
2. **bittensor-aml-subnet**: Lightweight Bittensor integration

Miner Flow

```
Miner Template API (port 8000)
  ↑
  | HTTP calls
  |
Subnet Miner (Bittensor neuron) ← Validator queries
```

Validator Flow

```
Validator queries all miners
  ↓
Collects submissions
  ↓
Strategy 2: Weighted Ensemble
  ↓
Canonical scores published
```

This architecture maximizes **reusability** (template), **simplicity** (proxy pattern), and **transparency** (all open source).