Machine Learning Recommendation Engine Architecture

Cryptocurrency Mining Monitoring System

Table of Contents

1. Overview

1.1 Purpose

The ML recommendation engine is a core component of the cryptocurrency mining monitoring system, designed to analyze real-time and historical data from mining operations to provide actionable insights and optimization recommendations. By leveraging machine learning algorithms and Abacus.AI's capabilities, the recommendation engine aims to maximize mining profitability, optimize resource utilization, and extend hardware lifespan through data-driven decision making.

The engine processes multiple data streams, including miner telemetry from Vnish firmware, pool performance data from Prohashing.com, and cryptocurrency market data from external APIs. It then generates personalized recommendations tailored to each user's specific mining setup, operational constraints, and financial goals.

1.2 Goals and Objectives

The recommendation engine aims to achieve the following objectives:

1. **Maximize Mining Profitability**: Provide recommendations that increase revenue and reduce operational costs, resulting in higher net profitability.
2. **Optimize Resource Utilization**: Suggest configurations that achieve the optimal balance between hashrate, power consumption, and hardware longevity.
3. **Reduce Operational Risks**: Identify potential issues before they lead to failures or significant performance degradation.
4. **Simplify Decision Making**: Transform complex data into clear, actionable recommendations that don't require deep technical expertise to implement.
5. **Enable Adaptive Operations**: Help mining operations adapt quickly to changing market conditions, network difficulties, and hardware performance.
6. **Personalize Recommendations**: Tailor suggestions based on each user's specific hardware, operational constraints, risk tolerance, and financial goals.
7. **Provide Continuous Improvement**: Learn from outcomes and user feedback to continuously improve recommendation quality and relevance.

1.3 System Context

The recommendation engine operates within the broader cryptocurrency mining monitoring system, which includes:

1. **Data Collection Layer**: Gathers telemetry from miners, performance data from mining pools, and market information from cryptocurrency APIs.
2. **Data Processing Pipeline**: Transforms, normalizes, and enriches raw data into structured formats suitable for analysis and model training.
3. **Feature Store**: Maintains a repository of features derived from the processed data, accessible for both batch and real-time inference.
4. **ML Models**: Trained algorithms that analyze patterns in the data to generate predictions and recommendations.
5. **Web Application**: Presents recommendations to users through intuitive dashboards and notifications.
6. **Feedback Collection**: Captures user actions and outcomes to improve future recommendations.

The recommendation engine interacts with these components to deliver value to users through the web application interface, while continuously improving through feedback loops and automated retraining.

2. Recommendation Types

The recommendation engine will generate several types of recommendations, each addressing different aspects of cryptocurrency mining operations.

2.1 Coin Switching Recommendations

These recommendations suggest optimal cryptocurrencies to mine based on profitability analysis, considering the user's hardware capabilities and merge mining opportunities.

Key Features:
- **Real-time Profitability Analysis**: Continuously evaluate the profitability of different coins based on current market prices, network difficulties, and pool rewards.
- **Merge Mining Optimization**: Identify optimal combinations of primary and auxiliary chains for merge mining to maximize total rewards.
- **Switching Cost Consideration**: Account for pool fees, setup time, and potential stability

issues when recommending coin switches.
- **Profitability Forecasting**: Predict short-term profitability trends to avoid frequent switching due to temporary market fluctuations.

Example Recommendation:

```
Switch from mining Bitcoin to Litecoin + Dogecoin (merge mining) for
the next 48 hours.
Expected profitability increase: 12.3%
Confidence: High (85%)
Reasoning: Recent Dogecoin price surge combined with stable Litecoin di
fficulty.
```

2.2 Power Optimization Recommendations

These recommendations suggest optimal power settings to maximize mining efficiency based on electricity costs, hardware capabilities, and market conditions.

Key Features:
- **Efficiency Curve Analysis**: Determine the optimal point on the power-hashrate curve for each miner based on its specific characteristics.
- **Dynamic Power Adjustments**: Recommend power adjustments based on time-of-day electricity pricing, market conditions, and thermal constraints.
- **Undervolting Guidance**: Provide safe undervolting recommendations to reduce power consumption while maintaining stability.
- **ROI-based Optimization**: Balance immediate profitability against hardware longevity for optimal long-term returns.

Example Recommendation:

```
Reduce power limit on Antminer S19 #003 from 3400W to 3100W.
Expected efficiency improvement: +8.5% (J/TH)
Expected hashrate impact: -3.2%
Net profitability impact: +4.1%
Confidence: Medium (75%)
```

2.3 Hardware Configuration Recommendations

These recommendations suggest optimal firmware settings, overclocking profiles, and cooling configurations to balance performance, efficiency, and hardware longevity.

Key Features:
- **Firmware Optimization**: Recommend optimal firmware settings based on hardware model, batch, and observed performance characteristics.
- **Thermal Management**: Suggest fan speed and cooling adjustments based on ambient temperature, miner workload, and thermal patterns.
- **Overclocking Profiles**: Provide personalized overclocking recommendations based on hardware capabilities and risk tolerance.
- **Stability Analysis**: Recommend configuration changes to improve operational stability and reduce error rates.

Example Recommendation:

```
Apply "Balanced" overclocking profile to Antminer S19 #001-#005.
Expected hashrate increase: +5.8%
Expected power increase: +3.2%
Net efficiency improvement: +2.5%
Stability impact: Minimal
Confidence: High (90%)
```

2.4 Maintenance Recommendations

These recommendations suggest preventive maintenance actions based on performance degradation patterns, error rates, and component lifecycle analysis.

Key Features:
- **Predictive Maintenance**: Identify miners that show early signs of potential failures before they occur.
- **Cleaning Schedules**: Recommend optimal timing for dust removal and cleaning based on environmental factors and performance degradation.
- **Component Replacement**: Suggest proactive replacement of fans or other components showing signs of wear.
- **Troubleshooting Guidance**: Provide specific troubleshooting steps for miners experiencing issues.

Example Recommendation:

```
Schedule maintenance for Antminer S19 #007 within the next 72 hours.
Warning signs: Increasing chip temperature variance, 12% hashrate de-
cline over 5 days.
Recommended actions: Clean dust from heat sinks, check fan #2 (showing
irregular RPM).
Priority: Medium
Confidence: High (88%)
```

2.5 Hardware Upgrade Recommendations

These recommendations suggest hardware upgrades or replacements based on ROI analysis, considering current hardware efficiency, market conditions, and new equipment options.

Key Features:
- **ROI Analysis**: Calculate expected return on investment for hardware upgrades based on current and projected profitability.
- **Replacement Prioritization**: Identify which miners should be replaced first based on efficiency, age, and performance.
- **New Hardware Evaluation**: Analyze the potential impact of new mining hardware releases on the market.
- **Upgrade Timing**: Recommend optimal timing for hardware upgrades based on market cycles and equipment availability.

Example Recommendation:

```
Consider replacing Antminer S17 units with S19j Pro models.
Estimated ROI period: 9.2 months
Efficiency improvement: 42%
Current resale value of S17 units: ~$1,200 each
Confidence: Medium (70%)
```

3. ML Models and Algorithms

The recommendation engine will employ multiple specialized models, each designed to address specific aspects of mining optimization. These models will work in concert to generate comprehensive recommendations.

3.1 Profitability Prediction Models

Purpose: Predict the profitability of mining different cryptocurrencies over various time horizons to inform coin switching recommendations.

Algorithms:
1. **Gradient Boosting Decision Trees (XGBoost/LightGBM)**
- **Justification**: Excellent performance on tabular data with complex relationships between features. Handles non-linear patterns in market data effectively.
- **Application**: Short-term profitability prediction (24-48 hours) for immediate coin switching decisions.

1. **LSTM Neural Networks**
   - **Justification**: Captures temporal dependencies in time-series data, essential for modeling market trends and cyclical patterns.
   - **Application**: Medium-term profitability forecasting (3-7 days) to avoid excessive coin switching.

2. **Ensemble Methods**
   - **Justification**: Combines predictions from multiple models to improve accuracy and robustness.
   - **Application**: Aggregate predictions across different time horizons and market scenarios.

Key Features:
- Historical and current market prices
- Network difficulty trends
- Mining pool rewards and fees
- Hash rate distribution
- Transaction fee markets
- Merge mining opportunities
- Seasonal and cyclical patterns

3.2 Power Efficiency Optimization Models

Purpose: Determine optimal power settings and configurations to maximize efficiency (hashrate per watt) while considering electricity costs and hardware constraints.

Algorithms:
1. **Bayesian Optimization**
- **Justification**: Efficiently explores the parameter space to find optimal configurations with minimal testing.
- **Application**: Optimize power limits, voltage settings, and clock frequencies.

1. **Reinforcement Learning**
   - **Justification**: Learns optimal policies through trial and feedback, adapting to changing conditions.

- **Application**: Dynamic power management based on real-time efficiency and market conditions.

2. **Gaussian Process Regression**
   - **Justification**: Provides uncertainty estimates along with predictions, useful for risk-aware optimization.
   - **Application**: Model the relationship between power settings, environmental conditions, and mining performance.

Key Features:
- Power consumption curves
- Hashrate response to power changes
- Thermal characteristics
- Electricity cost structures (including time-of-use pricing)
- Hardware-specific efficiency profiles
- Ambient temperature and cooling effectiveness

3.3 Anomaly Detection Models

Purpose: Identify abnormal behavior in mining operations that may indicate hardware issues, configuration problems, or security concerns.

Algorithms:
1. **Isolation Forest**
- **Justification**: Efficiently detects outliers in high-dimensional data without requiring extensive training data.
- **Application**: Identify miners with abnormal performance metrics relative to their peers.

1. **Autoencoder Neural Networks**
   - **Justification**: Learns normal operating patterns and can detect subtle deviations across multiple parameters.
   - **Application**: Detect gradual performance degradation and early warning signs of hardware issues.

2. **LSTM-based Sequence Anomaly Detection**
   - **Justification**: Captures temporal patterns in operational data to identify anomalous sequences.
   - **Application**: Detect unusual patterns in hashrate, temperature, or power consumption over time.

Key Features:
- Temporal patterns in hashrate, temperature, and power
- Error rates and rejected shares

- Fan performance metrics
- Network connectivity patterns
- Comparison with similar hardware
- Historical baseline performance

3.4 Hardware Lifecycle Models

Purpose: Predict hardware longevity, maintenance needs, and optimal replacement timing based on operational data and market conditions.

Algorithms:
1. **Survival Analysis Models (Cox Proportional Hazards)**
- **Justification**: Specifically designed to model time-to-event data, ideal for component failure prediction.
- **Application**: Predict the remaining useful life of mining hardware and components.

1. **Random Forest Regression**
   - **Justification**: Handles complex relationships between operational factors and hardware degradation.
   - **Application**: Predict maintenance needs and performance degradation rates.

2. **Economic Optimization Models**
   - **Justification**: Combines technical and financial factors to optimize economic decisions.
   - **Application**: Determine optimal timing for hardware replacement based on ROI analysis.

Key Features:
- Hardware age and operating hours
- Thermal history (average and peak temperatures)
- Power cycling frequency
- Maintenance history
- Performance degradation trends
- Market value of current and new hardware
- Projected profitability trends

3.5 Algorithm Selection Justification

The selection of algorithms for each model type is based on the following considerations:

1. **Data Characteristics**:
   - Time-series nature of mining and market data
   - Mix of categorical and numerical features
   - Presence of cyclical patterns and seasonality
   - Varying time horizons for different recommendation types

2. **Performance Requirements**:
   - Need for both batch processing (daily optimization) and real-time inference
   - Balance between accuracy and computational efficiency
   - Ability to quantify uncertainty in predictions
   - Interpretability requirements for user trust

3. **Practical Considerations**:
   - Integration capabilities with Abacus.AI platform
   - Availability of training data for different model types
   - Ease of deployment and maintenance
   - Ability to incorporate domain knowledge

The combination of tree-based models, neural networks, and specialized algorithms provides a comprehensive approach that addresses the diverse requirements of cryptocurrency mining optimization while leveraging the strengths of each algorithm type.

4. Feature Engineering

Effective feature engineering is critical to the performance of the recommendation engine. This section outlines the key features derived from various data sources and their importance in generating accurate recommendations.

4.1 Miner Telemetry Features

Features derived from Vnish firmware API data, capturing the operational state and performance of mining hardware.

Raw Features:
- Hashrate (overall and per hashboard)
- Temperature readings (PCB, chip, ambient)
- Power consumption and voltage
- Fan speeds and status
- Error codes and frequencies
- Accepted/rejected share counts
- Uptime and operational status

Derived Features:
1. **Efficiency Metrics**:
- Energy efficiency (J/TH)

- Performance efficiency (% of rated hashrate)
- Thermal efficiency (hashrate per degree Celsius)

1. **Stability Indicators**:
   - Hashrate variance (short and long-term)
   - Temperature stability index
   - Error rate trends
   - Share acceptance ratio

2. **Health Indicators**:
   - Temperature deviation from baseline
   - Fan performance index
   - Hashboard performance consistency
   - Power draw stability

3. **Temporal Patterns**:
   - 24-hour performance cycles
   - Weekly performance patterns
   - Performance degradation rate
   - Recovery time after restarts

4.2 Pool Performance Features

Features derived from Prohashing.com API data, capturing mining pool performance and rewards.

Raw Features:
- Worker hashrate (reported and effective)
- Share submission statistics
- Earnings data (amount, currency, time period)
- Coins mined (primary and merge-mined)
- Profitability metrics
- Pool difficulty and status

Derived Features:
1. **Reward Efficiency**:
- Earnings per hashrate (USD/TH/day)
- Merge mining bonus percentage
- Fee-adjusted profitability
- Reward variance

1. **Pool Comparison Metrics**:
   - Relative pool efficiency

   - Payout reliability score
   - Difficulty adjustment advantage
   - MEV (Miner Extractable Value) opportunity score

2. **Coin-specific Metrics**:
   - Coin profitability ranking
   - Merge mining compatibility score
   - Difficulty trend indicators
   - Reward halving proximity

3. **Temporal Patterns**:
   - Time-of-day profitability patterns
   - Day-of-week profitability patterns
   - Profitability trend indicators
   - Difficulty cycle position

4.3 Market Data Features

Features derived from cryptocurrency market APIs (CoinGecko, CoinMarketCap), capturing price movements and market conditions.

Raw Features:
- Current prices in USD and BTC
- Market capitalization
- Trading volume (24h)
- Price change percentages (1h, 24h, 7d)
- Network metrics (hashrate, difficulty)
- Technical indicators (RSI, MA, etc.)

Derived Features:
1. **Price Dynamics**:
- Volatility indicators (various time windows)
- Momentum signals
- Support/resistance proximity
- Price trend strength

1. **Market Sentiment**:
   - Market dominance shifts
   - Volume trend indicators
   - Relative strength across coins
   - Correlation with major assets

2. **Mining Economics**:
   - Block reward value
   - Transaction fee contribution
   - Mining difficulty adjusted return
   - Profitability horizon estimates

3. **Predictive Indicators**:
   - Short-term price movement probability
   - Difficulty adjustment forecasts
   - Halving impact projections
   - Market cycle position indicators

4.4 Derived Features

Complex features created by combining data across multiple sources to capture relationships and patterns relevant to mining optimization.

1. **Cross-Source Efficiency Metrics**:
   - Real profit margin (considering actual power costs)
   - Hardware-specific coin affinity score
   - Operational cost per dollar earned
   - Risk-adjusted return metrics

2. **Optimization Opportunity Indicators**:
   - Power optimization potential score
   - Coin switching benefit estimate
   - Configuration improvement potential
   - Maintenance urgency score

3. **Risk Metrics**:
   - Hardware failure probability
   - Profitability volatility index
   - Operational risk score
   - Market exposure metrics

4. **Temporal Correlation Features**:
   - Market-difficulty correlation
   - Price-hashrate relationship
   - Profitability cycle identification
   - Seasonal pattern strength

4.5 Feature Selection and Importance

Feature selection will be performed using a combination of domain knowledge and data-driven approaches:

1. **Domain-Driven Selection**:
   - Expert-identified critical metrics for mining operations
   - Known relationships between operational parameters and outcomes
   - Industry benchmarks and standards

2. **Data-Driven Selection**:
   - Feature importance from tree-based models
   - Correlation analysis to identify redundant features
   - Principal Component Analysis for dimensionality reduction
   - Recursive Feature Elimination with cross-validation

3. **Model-Specific Selection**:
   - Different feature subsets for different recommendation types
   - Time-horizon appropriate features for various prediction tasks
   - Algorithm-specific feature engineering

4. **Feature Importance Monitoring**:
   - Continuous evaluation of feature contribution to model performance
   - Drift detection in feature distributions
   - Periodic reassessment of feature relevance

The feature engineering process will be iterative, with new features being developed and tested based on model performance and user feedback. All features will be documented in a central feature registry within the Abacus.AI feature store to ensure consistency and reusability.

5. Abacus.AI Integration

The recommendation engine will leverage Abacus.AI's machine learning platform capabilities to streamline development, deployment, and management of ML models. This section outlines the integration approach with Abacus.AI's feature store, training pipelines, and inference services.

5.1 Feature Store Configuration

Abacus.AI's feature store will serve as the central repository for all features used by the recommendation engine, ensuring consistency between training and inference.

Feature Groups:

1. **Miner Telemetry Features**
   - **Entity ID**: `miner_id`
   - **Timestamp Column**: `timestamp`
   - **Features**: Hashrate, temperature, power, efficiency metrics, stability indicators
   - **Update Frequency**: 5-15 minutes (real-time stream)

2. **Pool Performance Features**
   - **Entity ID**: `worker_id`
   - **Timestamp Column**: `timestamp`
   - **Features**: Earnings, shares, profitability metrics, coin-specific performance
   - **Update Frequency**: 15-30 minutes (real-time stream)

3. **Market Data Features**
   - **Entity ID**: `coin_id`
   - **Timestamp Column**: `timestamp`
   - **Features**: Price, volume, market cap, technical indicators
   - **Update Frequency**: 5-15 minutes (real-time stream)

4. **Derived Mining Metrics**
   - **Entity ID**: `miner_id`
   - **Timestamp Column**: `timestamp`
   - **Features**: Cross-source metrics, optimization indicators, risk metrics
   - **Update Frequency**: 30-60 minutes (computed)

Feature Store Implementation:

1. **Data Ingestion**:
   - Configure real-time data streams from the data pipeline to Abacus.AI
   - Set up batch ingestion for historical data loading
   - Implement data validation rules to ensure quality

2. **Feature Computation**:
   - Define SQL transformations for simple derived features
   - Implement Python UDFs for complex feature computation
   - Schedule regular feature computation jobs

3. **Feature Serving**:
   - Configure online feature serving for real-time recommendations
   - Set up batch feature retrieval for model training
   - Implement caching strategies for frequently accessed features

4. **Feature Monitoring**:
   - Configure drift detection for critical features
   - Set up alerts for data quality issues
   - Implement feature validation tests

5.2 Model Training Automation

Abacus.AI's training pipelines will be used to automate the end-to-end process of model development, from data preparation to deployment.

Training Pipeline Configuration:

1. **Data Preparation**:
   - Feature selection based on recommendation type
   - Point-in-time feature joining for historical analysis
   - Train/validation/test splitting strategies
   - Data balancing and normalization

2. **Model Configuration**:
   - Hyperparameter optimization settings
   - Model architecture specifications
   - Training resource allocation
   - Evaluation metric definitions

3. **Training Schedule**:
   - Regular retraining intervals (daily/weekly)
   - Event-triggered retraining (data drift detection)
   - A/B test model training
   - Continuous training for online learning models

4. **Model Registry**:
   - Version control for all trained models
   - Model metadata and performance tracking
   - Approval workflow for production deployment
   - Rollback capabilities for problematic models

Model Types and Training Configurations:

1. **Profitability Prediction Models**:
   - Training frequency: Daily
   - Lookback period: 90 days
   - Validation strategy: Time-based cross-validation
   - Key metrics: RMSE, MAE, directional accuracy

2. **Power Optimization Models**:
   - Training frequency: Weekly
   - Lookback period: 30 days
   - Validation strategy: Hardware-stratified cross-validation
   - Key metrics: Efficiency improvement, stability impact

3. **Anomaly Detection Models**:
   - Training frequency: Weekly
   - Lookback period: 60 days
   - Validation strategy: Precision-recall evaluation
   - Key metrics: F1 score, detection latency, false positive rate

4. **Hardware Lifecycle Models**:
   - Training frequency: Monthly
   - Lookback period: Full hardware history
   - Validation strategy: Survival analysis metrics
   - Key metrics: C-index, calibration error, economic impact

5.3 Real-time Inference

Abacus.AI's real-time inference capabilities will be leveraged to generate recommendations based on the latest data.

Inference Service Configuration:

1. **Endpoint Setup**:
   - Dedicated endpoints for each recommendation type
   - Scalable infrastructure for varying load
   - Low-latency configuration for real-time responses
   - Authentication and access control

2. **Feature Retrieval**:
   - Real-time feature serving from feature store
   - Point-in-time lookups for historical context
   - Feature vector assembly for model input
   - Caching strategies for performance optimization

3. **Recommendation Generation**:
   - Model inference with latest features
   - Post-processing of model outputs
   - Confidence scoring and filtering
   - Recommendation formatting and enrichment

4. **Response Handling**:
   - Structured recommendation delivery
   - Explanation generation for transparency
   - Fallback strategies for inference failures
   - Performance logging and monitoring

Real-time Recommendation Workflow:

1. User accesses dashboard or triggers recommendation request
2. System retrieves latest features from feature store
3. Models generate predictions and recommendation candidates
4. Recommendations are filtered, ranked, and enriched with explanations
5. Final recommendations are delivered to the user interface
6. User actions and feedback are captured for continuous improvement

5.4 Model Monitoring and Management

Abacus.AI's model monitoring capabilities will be used to ensure the ongoing performance and reliability of the recommendation engine.

Monitoring Configuration:

1. **Performance Monitoring**:
   - Track key metrics for each model type
   - Compare against baseline and historical performance
   - Alert on significant performance degradation
   - Visualize performance trends over time

2. **Data Drift Detection**:
   - Monitor input feature distributions
   - Detect concept drift in target variables
   - Identify feature correlation changes
   - Trigger retraining when significant drift occurs

3. **Operational Monitoring**:
   - Track inference latency and throughput
   - Monitor resource utilization
   - Log error rates and failure modes
   - Alert on service degradation

4. **Feedback Monitoring**:
   - Track recommendation acceptance rates

- Measure actual vs. predicted outcomes
- Analyze user feedback patterns
- Identify systematic recommendation issues

Model Management Workflow:

1. **Continuous Evaluation**:
   - Regular performance assessment against fresh data
   - Comparison with champion models
   - Evaluation of business impact metrics
   - User satisfaction analysis

2. **Model Updates**:
   - Scheduled retraining based on performance
   - Champion/challenger testing for new models
   - Gradual rollout of model updates
   - Automated fallback to previous versions if issues detected

3. **Model Governance**:
   - Documentation of model versions and changes
   - Tracking of training data lineage
   - Audit trail of deployment decisions
   - Performance history preservation

6. Training and Inference Pipelines

This section details the end-to-end pipelines for data preparation, model training, and recommendation generation, ensuring a systematic approach to developing and deploying the recommendation engine.

6.1 Data Preparation Pipeline

The data preparation pipeline transforms raw data from various sources into feature sets suitable for model training and inference.

Pipeline Stages:

1. **Data Collection**:
   - Retrieve data from Vnish firmware API, Prohashing API, and market data APIs
   - Validate data completeness and format
   - Handle missing or delayed data sources

2. **Data Cleaning**:
   - Remove invalid or corrupted records
   - Handle missing values through imputation or flagging
   - Filter out outliers and anomalous data points
   - Normalize units and formats across sources

3. **Feature Extraction**:
   - Calculate basic derived features from raw data
   - Generate time-window aggregations (hourly, daily, weekly)
   - Compute statistical metrics (mean, variance, trends)
   - Extract domain-specific indicators

4. **Feature Transformation**:
   - Normalize numerical features
   - Encode categorical variables
   - Apply transformations for skewed distributions
   - Scale features to appropriate ranges

5. **Feature Selection**:
   - Filter features based on relevance to specific models
   - Remove highly correlated features
   - Select features based on importance scores
   - Create feature subsets for different recommendation types

6. **Dataset Creation**:
   - Join features across different sources
   - Create point-in-time training datasets
   - Generate labels for supervised learning tasks
   - Split data into training, validation, and test sets

Implementation:

```
# Pseudocode for data preparation pipeline
def prepare_data_for_training(start_date, end_date, model_type):
    # 1. Collect raw data
    miner_data = fetch_miner_telemetry(start_date, end_date)
    pool_data = fetch_pool_performance(start_date, end_date)
    market_data = fetch_market_data(start_date, end_date)

    # 2. Clean data
    miner_data = clean_miner_data(miner_data)
    pool_data = clean_pool_data(pool_data)
    market_data = clean_market_data(market_data)

    # 3. Extract features
    miner_features = extract_miner_features(miner_data)
    pool_features = extract_pool_features(pool_data)
    market_features = extract_market_features(market_data)

    # 4. Transform features
    miner_features = transform_features(miner_features)
    pool_features = transform_features(pool_features)
    market_features = transform_features(market_features)

    # 5. Select features based on model type
    selected_features = select_features_for_model(
        miner_features, pool_features, market_features, model_type
    )

    # 6. Create dataset with appropriate labels
    if model_type == "profitability_prediction":
        labels = generate_profitability_labels(pool_data, fu-
ture_window=24)
    elif model_type == "power_optimization":
        labels = generate_efficiency_labels(miner_data, pool_data)
    # ... other model types

    # Create final dataset
    dataset = create_training_dataset(selected_features, labels)
    train, val, test = split_dataset(dataset)

    return train, val, test
```

6.2 Model Training Pipeline

The model training pipeline handles the process of training, evaluating, and registering models for different recommendation types.

Pipeline Stages:

1. **Hyperparameter Optimization**:
   - Define hyperparameter search space
   - Implement search strategy (grid, random, Bayesian)
   - Evaluate configurations using cross-validation
   - Select optimal hyperparameters

2. **Model Training**:
   - Initialize model with optimal hyperparameters
   - Train on prepared dataset
   - Implement early stopping and regularization
   - Track training metrics and convergence

3. **Model Evaluation**:
   - Evaluate on holdout test set
   - Calculate performance metrics
   - Generate performance visualizations
   - Compare against baseline and previous versions

4. **Model Explanation**:
   - Generate feature importance analysis
   - Create partial dependence plots
   - Implement SHAP or LIME explanations
   - Document model behavior patterns

5. **Model Registration**:
   - Save model artifacts and metadata
   - Register model in Abacus.AI model registry
   - Document training process and results
   - Tag model with version and purpose

Implementation:

```
# Pseudocode for model training pipeline
def train_model(train_data, val_data, test_data, model_type):
    # 1. Hyperparameter optimization
    param_space = define_hyperparameter_space(model_type)
    best_params = optimize_hyperparameters(
        train_data, val_data, param_space, model_type
    )

    # 2. Model training
    model = initialize_model(model_type, best_params)
    training_history = model.fit(
        train_data,
        validation_data=val_data,
        early_stopping=True
    )

    # 3. Model evaluation
    metrics = evaluate_model(model, test_data)
    performance_report = generate_performance_report(metrics, train-
ing_history)

    # 4. Model explanation
    feature_importance = calculate_feature_importance(model, test_data)
    explanations = generate_model_explanations(model, test_data)

    # 5. Model registration
    model_metadata = {
        "type": model_type,
        "version": generate_version(),
        "parameters": best_params,
        "performance": metrics,
        "training_date": current_timestamp(),
        "feature_importance": feature_importance
    }

    model_id = register_model(model, model_metadata)

    return model_id, performance_report
```

6.3 Real-time Inference Pipeline

The real-time inference pipeline generates recommendations on demand based on the latest data and user context.

Pipeline Stages:

1. **Request Processing**:
   - Parse recommendation request
   - Validate request parameters
   - Identify user context and preferences
   - Determine recommendation types needed

2. **Feature Retrieval**:
   - Fetch latest features from feature store
   - Retrieve historical context as needed
   - Assemble feature vectors for each model
   - Apply necessary transformations

3. **Model Inference**:
   - Load appropriate models from registry
   - Generate predictions for each recommendation type
   - Calculate prediction confidence scores
   - Apply business rules and constraints

4. **Recommendation Generation**:
   - Filter and rank recommendation candidates
   - Apply user preferences and constraints
   - Generate explanations for recommendations
   - Format recommendations for presentation

5. **Response Delivery**:
   - Package recommendations in response format
   - Include metadata and confidence scores
   - Log recommendation details for feedback
   - Return response to requesting application

Implementation:

```
# Pseudocode for real-time inference pipeline
def generate_recommendations(user_id, context):
    # 1. Process request
    user_preferences = get_user_preferences(user_id)
    recommendation_types = determine_recommendation_types(context)

    # 2. Retrieve features
    miner_features = get_latest_miner_features(user_id)
    pool_features = get_latest_pool_features(user_id)
    market_features = get_latest_market_features()

    recommendations = []

    # 3 & 4. Model inference and recommendation generation
    for rec_type in recommendation_types:
        # Get appropriate model
        model = load_model(rec_type)

        # Prepare features for this model
        features = prepare_features_for_model(
            miner_features, pool_features, market_features, rec_type
        )

        # Generate predictions
        predictions = model.predict(features)

        # Apply business rules
        filtered_predictions = apply_business_rules(
            predictions, user_preferences, context
        )

        # Generate recommendations
        rec_candidates = generate_recommendation_candidates(
            filtered_predictions, rec_type
        )

        # Add explanations
        rec_candidates = add_explanations(rec_candidates, model, fea-
tures)

        recommendations.extend(rec_candidates)

    # Rank across recommendation types if needed
    final_recommendations = rank_recommendations(recommendations,
user_preferences)

    # 5. Prepare response
    response = format_recommendations(final_recommendations)
```

```
    log_recommendations(user_id, final_recommendations, context)

    return response
```

6.4 Batch Inference Pipeline

The batch inference pipeline generates recommendations periodically for all users, enabling scheduled updates and notifications.

Pipeline Stages:

1. **User Enumeration**:
   - Identify all active users
   - Group users by mining setup similarity
   - Prioritize users based on activity and value
   - Determine recommendation types for each user

2. **Batch Feature Preparation**:
   - Retrieve features for all users
   - Prepare feature matrices for efficient processing
   - Apply necessary transformations
   - Handle missing or incomplete data

3. **Batch Prediction**:
   - Load models for each recommendation type
   - Generate predictions for all users
   - Parallelize prediction across user groups
   - Track prediction metadata and timing

4. **Recommendation Processing**:
   - Apply user-specific filters and constraints
   - Generate explanations for each recommendation
   - Prioritize recommendations by impact and confidence
   - Format for storage and delivery

5. **Storage and Notification**:
   - Store recommendations in database
   - Determine notification urgency and channels
   - Generate notification content
   - Schedule delivery based on user preferences

Implementation:

```
# Pseudocode for batch inference pipeline
def batch_generate_recommendations():
    # 1. Enumerate users
    active_users = get_active_users()
    user_groups = group_users_by_similarity(active_users)

    all_recommendations = {}

    # 2, 3, & 4. Feature preparation, prediction, and processing
    for group in user_groups:
        # Prepare features for this group
        group_features = prepare_batch_features(group)

        for rec_type in RECOMMENDATION_TYPES:
            # Load model
            model = load_model(rec_type)

            # Generate predictions
            predictions = model.predict_batch(group_features)

            # Process into recommendations
            for user_id in group:
                user_predictions = predictions[user_id]
                user_preferences = get_user_preferences(user_id)

                # Apply filters and generate recommendations
                user_recs = process_user_recommendations(
                    user_predictions, user_preferences, rec_type
                )

                # Add to user's recommendations
                if user_id not in all_recommendations:
                    all_recommendations[user_id] = []
                all_recommendations[user_id].extend(user_recs)

    # 5. Storage and notification
    for user_id, recommendations in all_recommendations.items():
        # Prioritize recommendations
        prioritized_recs = prioritize_recommendations(recommendations)

        # Store in database
        store_user_recommendations(user_id, prioritized_recs)

        # Determine notification needs
        urgent_recs = [r for r in prioritized_recs if r['urgency'] == '
high']
        if urgent_recs:
            send_notification(user_id, urgent_recs)
```

```
    return len(all_recommendations)
```

7. Recommendation Evaluation

Evaluating the quality and impact of recommendations is essential for continuous improvement of the recommendation engine. This section outlines the evaluation framework, metrics, and feedback mechanisms.

7.1 Evaluation Metrics

Different types of recommendations require different evaluation metrics to assess their effectiveness.

Profitability Recommendation Metrics:

1. **Accuracy Metrics**:
   - Mean Absolute Error (MAE) of profitability predictions
   - Root Mean Squared Error (RMSE) of revenue forecasts
   - Directional accuracy (correct prediction of profitability increases/decreases)
   - Precision and recall for profitable coin switching recommendations

2. **Economic Impact Metrics**:
   - Realized profit improvement percentage
   - Opportunity cost of missed switches
   - Risk-adjusted return improvement
   - Cumulative profit delta over baseline strategy

Power Optimization Metrics:

1. **Efficiency Metrics**:
   - Energy efficiency improvement (J/TH)
   - Power cost reduction percentage
   - Hashrate retention ratio (maintained hashrate / original hashrate)
   - Efficiency-adjusted profit improvement

2. **Operational Metrics**:
   - Stability impact score (error rate changes)
   - Temperature reduction achieved
   - Hardware stress reduction estimate
   - Implementation success rate

Hardware Recommendation Metrics:

1. **Maintenance Metrics**:
   - Failure prediction accuracy
   - Preventive maintenance effectiveness
   - Downtime reduction percentage
   - Mean time between failures improvement

2. **Upgrade Metrics**:
   - ROI prediction accuracy
   - Actual vs. predicted payback period
   - Upgrade timing optimality score
   - Long-term profitability impact

Overall Recommendation Metrics:

1. **User Engagement Metrics**:
   - Recommendation acceptance rate
   - Implementation completion rate
   - Time to implementation
   - User satisfaction scores

2. **System Performance Metrics**:
   - Recommendation generation latency
   - Feature freshness at inference time
   - Model drift detection effectiveness
   - System reliability and uptime

7.2 A/B Testing Framework

A systematic approach to testing new recommendation models and strategies against existing ones.

Testing Methodology:

1. **Test Design**:
   - Define test hypotheses and success criteria
   - Determine appropriate sample sizes and duration
   - Select user segments for testing
   - Design control and treatment groups

2. **Implementation**:
   - Deploy champion and challenger models simultaneously
   - Randomly assign users to test groups

   - Ensure consistent feature availability across groups
   - Monitor test execution and data collection

3. **Analysis**:
   - Calculate key performance metrics for each group
   - Perform statistical significance testing
   - Analyze subgroup performance differences
   - Identify unexpected outcomes or side effects

4. **Decision Making**:
   - Establish clear criteria for test success
   - Define procedures for promoting challengers to champions
   - Document test results and decisions
   - Plan follow-up tests based on findings

A/B Testing Pipeline:

```
# Pseudocode for A/B testing framework
def setup_ab_test(test_name, hypothesis, champion_model, chal-
lenger_model, duration_days):
    # Create test configuration
    test_config = {
        "name": test_name,
        "hypothesis": hypothesis,
        "champion_id": champion_model.id,
        "challenger_id": challenger_model.id,
        "start_date": current_date(),
        "end_date": current_date() + timedelta(days=duration_days),
        "metrics": define_test_metrics(),
        "success_criteria": define_success_criteria()
    }

    # Select user groups
    eligible_users = get_eligible_users_for_test()
    control_group, treatment_group = ran-
domly_split_users(eligible_users)

    test_config["control_group"] = control_group
    test_config["treatment_group"] = treatment_group

    # Register test
    test_id = register_ab_test(test_config)

    # Configure routing
    configure_model_routing(
        control_group, champion_model.id,
        treatment_group, challenger_model.id
    )

    return test_id

def analyze_ab_test(test_id):
    # Retrieve test configuration
    test_config = get_test_config(test_id)

    # Collect performance data
    control_data = get_performance_data(
        test_config["control_group"],
        test_config["start_date"],
        test_config["end_date"]
    )

    treatment_data = get_performance_data(
        test_config["treatment_group"],
        test_config["start_date"],
```

```
        test_config["end_date"]
    )

    # Calculate metrics
    control_metrics = calculate_metrics(control_data, test_config["met-
rics"])
    treatment_metrics = calculate_metrics(treatment_data, test_config["
metrics"])

    # Perform statistical analysis
    analysis_results = statistical_analysis(control_metrics, treat-
ment_metrics)

    # Evaluate success criteria
    success_evaluation = evaluate_success_criteria(
        analysis_results, test_config["success_criteria"]
    )

    # Generate report
    report = generate_ab_test_report(
        test_config, control_metrics, treatment_metrics,
        analysis_results, success_evaluation
    )

    return report, success_evaluation
```

7.3 User Feedback Collection

Mechanisms for collecting explicit and implicit feedback from users to improve recommendation quality.

Feedback Types:

1. **Explicit Feedback**:
   - Recommendation acceptance/rejection
   - Implementation success reporting
   - Satisfaction ratings
   - Free-form comments and suggestions

2. **Implicit Feedback**:
   - Recommendation view time
   - Detail expansion actions
   - Implementation initiation
   - Follow-up action timing

Collection Methods:

1. **In-App Feedback**:
   - Simple accept/reject buttons for recommendations
   - Implementation outcome reporting
   - Quick satisfaction surveys
   - Feedback forms for detailed input

2. **Behavioral Tracking**:
   - User interaction logging
   - Implementation action tracking
   - Configuration change monitoring
   - Result comparison before/after implementation

3. **Outcome Measurement**:
   - Automatic verification of implementation when possible
   - Performance measurement after changes
   - Comparison of actual vs. predicted outcomes
   - Long-term impact assessment

Feedback Integration:

```
# Pseudocode for feedback collection and integration
def collect_explicit_feedback(user_id, recommendation_id, feed-
back_type, details=None):
    # Record the feedback
    feedback = {
        "user_id": user_id,
        "recommendation_id": recommendation_id,
        "timestamp": current_timestamp(),
        "feedback_type": feedback_type,
        "details": details
    }

    store_feedback(feedback)

    # Update recommendation status
    update_recommendation_status(recommendation_id, feedback_type)

    # Trigger immediate learning if appropriate
    if feedback_type in ["rejected_incorrect", "implemen-
ted_unsuccessful"]:
        trigger_feedback_analysis(recommendation_id, feedback)

    return feedback_id

def track_implicit_feedback(user_id, recommendation_id, action_type):
    # Record the interaction
    interaction = {
        "user_id": user_id,
        "recommendation_id": recommendation_id,
        "timestamp": current_timestamp(),
        "action_type": action_type
    }

    store_interaction(interaction)

    # Update engagement metrics
    update_engagement_metrics(user_id, recommendation_id, action_type)

    return interaction_id

def verify_recommendation_outcome(recommendation_id):
    # Get recommendation details
    recommendation = get_recommendation(recommendation_id)

    # Get relevant metrics before and after implementation
    before_metrics = get_metrics_before_implementation(recommendation)
    after_metrics = get_current_metrics(recommendation)
```

```
    # Calculate impact
    impact = calculate_recommendation_impact(before_metrics,
after_metrics)

    # Compare with predicted impact
    accuracy = compare_actual_vs_predicted(impact, recommendation["pre-
dicted_impact"])

    # Store outcome verification
    store_outcome_verification(recommendation_id, impact, accuracy)

    # Update model feedback
    update_model_feedback(recommendation["model_id"], accuracy)

    return impact, accuracy
```

7.4 Performance Monitoring

Continuous monitoring of recommendation engine performance to identify issues and improvement opportunities.

Monitoring Dimensions:

1. **Model Performance**:
   - Prediction accuracy over time
   - Feature importance stability
   - Model drift indicators
   - Performance across user segments

2. **Recommendation Quality**:
   - Acceptance rate trends
   - Implementation success rate
   - Economic impact measurements
   - User satisfaction metrics

3. **System Performance**:
   - Inference latency statistics
   - Feature freshness metrics
   - Pipeline reliability measures
   - Resource utilization patterns

4. **Business Impact**:
   - Overall profitability improvement
   - User retention and engagement

- Feature adoption metrics
- Return on investment measures

Monitoring Implementation:

```
# Pseudocode for performance monitoring
def setup_performance_monitoring():
    # Define monitoring metrics
    model_metrics = define_model_performance_metrics()
    recommendation_metrics = define_recommendation_quality_metrics()
    system_metrics = define_system_performance_metrics()
    business_metrics = define_business_impact_metrics()

    # Configure data collection
    configure_metrics_collection(model_metrics)
    configure_metrics_collection(recommendation_metrics)
    configure_metrics_collection(system_metrics)
    configure_metrics_collection(business_metrics)

    # Set up dashboards
    create_monitoring_dashboard(model_metrics, "Model Performance")
    create_monitoring_dashboard(recommendation_metrics,
"Recommendation Quality")
    create_monitoring_dashboard(system_metrics, "System Performance")
    create_monitoring_dashboard(business_metrics, "Business Impact")

    # Configure alerts
    setup_performance_alerts(model_metrics)
    setup_performance_alerts(recommendation_metrics)
    setup_performance_alerts(system_metrics)
    setup_performance_alerts(business_metrics)

    return monitoring_config_id

def generate_performance_report(start_date, end_date):
    # Collect metrics for the period
    model_performance = collect_metrics("model_performance",
start_date, end_date)
    recommendation_quality = collect_metrics("recommendation_quality",
start_date, end_date)
    system_performance = collect_metrics("system_performance",
start_date, end_date)
    business_impact = collect_metrics("business_impact", start_date,
end_date)

    # Analyze trends
    model_trends = analyze_metric_trends(model_performance)
    recommendation_trends = ana-
lyze_metric_trends(recommendation_quality)
    system_trends = analyze_metric_trends(system_performance)
    business_trends = analyze_metric_trends(business_impact)

    # Generate insights
```

```
    insights = generate_performance_insights(
        model_trends, recommendation_trends, system_trends,
business_trends
    )

    # Create report
    report = create_performance_report(
        start_date, end_date,
        model_performance, recommendation_quality,
        system_performance, business_impact,
        insights
    )

    return report
```

8. Continuous Learning

The recommendation engine will continuously improve through automated learning from new data, user feedback, and changing conditions. This section outlines the approach to maintaining and enhancing model performance over time.

8.1 Model Retraining Strategy

A systematic approach to updating models with new data and insights.

Retraining Triggers:

1. **Scheduled Retraining**:
   - Regular intervals based on model type and data velocity
   - Daily updates for market-sensitive models
   - Weekly updates for operational models
   - Monthly updates for hardware lifecycle models

2. **Event-Based Retraining**:
   - Significant market events (price movements, halving events)
   - Detection of data drift beyond thresholds
   - Performance degradation below acceptable levels
   - New feature availability or significant feature changes

3. **Feedback-Driven Retraining**:
   - Accumulation of sufficient new feedback data
   - Detection of systematic errors in recommendations

- Significant changes in user behavior or preferences
- New edge cases or failure modes identified

Retraining Process:

```
# Pseudocode for model retraining strategy
def evaluate_retraining_need(model_id):
    # Get model metadata
    model_metadata = get_model_metadata(model_id)
    model_type = model_metadata["type"]
    last_training_date = model_metadata["training_date"]

    # Check scheduled retraining
    if is_scheduled_retraining_due(model_type, last_training_date):
        return True, "scheduled_retraining"

    # Check performance metrics
    current_performance = get_current_performance(model_id)
    if performance_degraded(current_performance, model_metadata["baseli
ne_performance"]):
        return True, "performance_degradation"

    # Check data drift
    drift_metrics = calculate_data_drift(model_id)
    if drift_detected(drift_metrics):
        return True, "data_drift"

    # Check feedback signals
    feedback_metrics = analyze_recent_feedback(model_id)
    if feedback_indicates_retraining(feedback_metrics):
        return True, "feedback_signals"

    return False, None

def retrain_model(model_id, reason):
    # Get model metadata
    model_metadata = get_model_metadata(model_id)
    model_type = model_metadata["type"]

    # Determine training parameters
    if reason == "data_drift":
        # Focus on recent data
        training_params = get_drift_adaptation_params(model_type)
    elif reason == "performance_degradation":
        # Focus on problematic cases
        training_params = get_performance_recovery_params(model_type)
    elif reason == "feedback_signals":
        # Incorporate feedback data
        training_params = get_feedback_integration_params(model_type)
    else:
        # Standard retraining
        training_params = get_standard_training_params(model_type)
```

```
    # Prepare training data
    train_data, val_data, test_data = prepare_training_data(model_type,
training_params)

    # Train new model version
    new_model_id = train_model(train_data, val_data, test_data,
model_type)

    # Evaluate new model
    evaluation_results = evaluate_model_upgrade(model_id, new_model_id,
test_data)

    # Deploy if improved
    if evaluation_results["should_deploy"]:
        deploy_model(new_model_id, replace_model_id=model_id)
        log_model_upgrade(model_id, new_model_id, reason,
evaluation_results)
        return new_model_id
    else:
        log_failed_upgrade(model_id, new_model_id, reason,
evaluation_results)
        return model_id
```

8.2 Feedback Loop Integration

Mechanisms for incorporating user feedback into the model improvement process.

Feedback Integration Methods:

1. **Direct Model Updates**:
   - Immediate correction of clear errors
   - Online learning for incremental model updates
   - Reinforcement learning from user actions
   - Active learning for ambiguous cases

2. **Training Data Enhancement**:
   - Augmentation of training data with feedback cases
   - Weighting recent feedback more heavily
   - Creating synthetic examples from feedback patterns
   - Correcting mislabeled training instances

3. **Feature Engineering Improvements**:
   - Identifying missing features from feedback analysis
   - Adjusting feature importance based on feedback

- Creating new derived features to address gaps
- Removing or downweighting misleading features

4. **Recommendation Post-Processing**:
   - Adjusting confidence scores based on feedback history
   - Implementing business rules derived from feedback patterns
   - Personalizing recommendation presentation based on user preferences
   - Filtering recommendations with high rejection probability

Feedback Loop Implementation:

```
# Pseudocode for feedback loop integration
def process_recommendation_feedback(recommendation_id, feedback):
    # Get recommendation details
    recommendation = get_recommendation(recommendation_id)
    model_id = recommendation["model_id"]

    # Record feedback for future training
    store_labeled_example(
        model_id,
        recommendation["features"],
        recommendation["prediction"],
        feedback["actual_outcome"]
    )

    # Analyze feedback for immediate learning
    if feedback["feedback_type"] in ["rejected_incorrect", "implemen-
ted_unsuccessful"]:
        # Identify potential issues
        issue_analysis = analyze_recommendation_issue(recommendation,
feedback)

        # Update model if clear correction is possible
        if issue_analysis["clear_correction"]:
            apply_model_correction(model_id, issue_analysis["correc-
tion"])

        # Update business rules if pattern detected
        if issue_analysis["rule_candidate"]:
            evaluate_and_update_business_rules(issue_analysis["rule_can
didate"])

        # Flag for feature engineering if feature gap identified
        if issue_analysis["feature_gap"]:
            register_feature_engineering_task(issue_analysis["fea-
ture_gap"])

    # Update user preference model
    update_user_preferences(
        recommendation["user_id"],
        recommendation["type"],
        feedback["feedback_type"]
    )

    # Check if retraining threshold reached
    feedback_stats = get_feedback_statistics(model_id)
    if feedback_stats["negative_feedback_rate"] > RETRAINING_THRESHOLD:
        trigger_model_retraining(model_id, "feed-
back_threshold_exceeded")
```

```
    return issue_analysis

def incorporate_feedback_in_training(model_type, training_data):
    # Get relevant feedback data
    feedback_data = get_feedback_training_data(model_type)

    # Weight feedback data appropriately
    weighted_feedback = apply_feedback_weighting(feedback_data)

    # Combine with standard training data
    enhanced_training_data = combine_training_data(training_data,
weighted_feedback)

    # Apply data balancing if needed
    if data_imbalance_detected(enhanced_training_data):
        enhanced_training_data = bal-
ance_training_data(enhanced_training_data)

    return enhanced_training_data
```

8.3 Concept Drift Detection

Methods for identifying when models become less effective due to changing patterns in the underlying data.

Drift Detection Approaches:

1. **Statistical Monitoring**:
   - Track feature distribution changes over time
   - Monitor prediction distribution shifts
   - Measure input-output relationship stability
   - Calculate statistical distance metrics between time periods

2. **Performance-Based Detection**:
   - Monitor error metrics on recent data
   - Compare performance across different user segments
   - Track recommendation acceptance rate trends
   - Measure economic impact of recommendations over time

3. **Feedback-Based Detection**:
   - Analyze patterns in user feedback
   - Monitor rejection reasons and categories
   - Track implementation success rate changes
   - Identify emerging failure modes

4. **Domain-Specific Indicators**:
   - Monitor cryptocurrency market regime changes
   - Track mining hardware technology evolution
   - Identify regulatory or environmental policy shifts
   - Detect changes in electricity pricing models

Drift Detection Implementation:

```
# Pseudocode for concept drift detection
def monitor_data_drift(model_id, window_size=30):
    # Get model metadata
    model_metadata = get_model_metadata(model_id)
    reference_distribution = model_metadata["training_distribution"]

    # Get recent feature data
    recent_data = get_recent_feature_data(model_id, days=window_size)

    # Calculate distribution metrics
    distribution_metrics = {}
    for feature in reference_distribution:
        reference = reference_distribution[feature]
        current = calculate_feature_distribution(recent_data, feature)

        # Calculate statistical distance
        if is_numerical_feature(feature):
            distance = calculate_ks_distance(reference, current)
        else:
            distance = calculate_js_divergence(reference, current)

        distribution_metrics[feature] = {
            "distance": distance,
            "reference": reference,
            "current": current
        }

    # Calculate overall drift score
    drift_score = calculate_overall_drift_score(distribution_metrics)

    # Check against threshold
    drift_detected = drift_score > DRIFT_THRESHOLD

    # Store drift monitoring results
    store_drift_monitoring_results(model_id, drift_score,
distribution_metrics)

    # Trigger alerts if needed
    if drift_detected:
        trigger_drift_alert(model_id, drift_score, distribu-
tion_metrics)
        evaluate_retraining_need(model_id)

    return drift_detected, drift_score, distribution_metrics

def monitor_performance_drift(model_id, window_size=30):
    # Get model metadata
    model_metadata = get_model_metadata(model_id)
```

```
    baseline_performance = model_metadata["baseline_performance"]

    # Get recent performance metrics
    recent_performance = get_recent_performance(model_id, days=win-
dow_size)

    # Calculate performance changes
    performance_changes = {}
    for metric in baseline_performance:
        baseline = baseline_performance[metric]
        current = recent_performance[metric]

        # Calculate relative change
        change = (current - baseline) / baseline

        performance_changes[metric] = {
            "baseline": baseline,
            "current": current,
            "change": change
        }

    # Calculate overall performance drift
    performance_drift = calcu-
late_performance_drift(performance_changes)

    # Check against threshold
    significant_drift = performance_drift > PERFORMANCE_DRIFT_THRESHOLD

    # Store monitoring results
    store_performance_monitoring_results(model_id, performance_drift,
performance_changes)

    # Trigger alerts if needed
    if significant_drift:
        trigger_performance_alert(model_id, performance_drift, perform-
ance_changes)
        evaluate_retraining_need(model_id)

    return significant_drift, performance_drift, performance_changes
```

8.4 Model Versioning and Rollback

Mechanisms for managing model versions and reverting to previous versions when necessary.

Versioning Strategy:

1. **Version Control**:
   - Unique identifiers for each model version
   - Comprehensive metadata for each version
   - Training data lineage tracking
   - Performance benchmark preservation

2. **Deployment Management**:
   - Controlled rollout of new versions
   - Canary testing before full deployment
   - A/B testing for major changes
   - Shadow mode evaluation for critical models

3. **Rollback Mechanisms**:
   - Automated performance monitoring with alerts
   - Quick rollback procedures for emergencies
   - Partial rollbacks for specific user segments
   - Gradual rollback for non-critical issues

4. **Version Lifecycle Management**:
   - Archiving of older versions
   - Retention policies for model artifacts
   - Documentation of version history
   - Audit trail of deployment decisions

Implementation:

```
# Pseudocode for model versioning and rollback
def create_model_version(model_type, model_artifact, train-
ing_metadata):
    # Generate version identifier
    version_id = generate_version_id(model_type)

    # Create version metadata
    version_metadata = {
        "model_type": model_type,
        "version_id": version_id,
        "created_at": current_timestamp(),
        "created_by": current_user(),
        "training_data": training_metadata["training_data_id"],
        "parameters": training_metadata["parameters"],
        "performance": training_metadata["performance"],
        "baseline_distribution": training_metadata["fea-
ture_distribution"],
        "status": "created"
    }

    # Store model artifact
    artifact_location = store_model_artifact(model_type, version_id,
model_artifact)
    version_metadata["artifact_location"] = artifact_location

    # Register version in registry
    register_model_version(version_metadata)

    return version_id

def deploy_model_version(version_id, deployment_strategy="canary"):
    # Get version metadata
    version_metadata = get_model_version(version_id)
    model_type = version_metadata["model_type"]

    # Get current production version
    current_version = get_production_version(model_type)

    # Update version status
    update_version_status(version_id, "deploying")

    if deployment_strategy == "full":
        # Full deployment
        set_production_version(model_type, version_id)
        update_version_status(current_version, "archived")
        update_version_status(version_id, "production")

    elif deployment_strategy == "canary":
```

```
        # Canary testing
        canary_config = {
            "model_type": model_type,
            "new_version": version_id,
            "current_version": current_version,
            "traffic_percentage": 10,
            "evaluation_period": 24,  # hours
            "success_criteria":
define_canary_success_criteria(model_type)
        }

        start_canary_test(canary_config)
        update_version_status(version_id, "canary_testing")

    elif deployment_strategy == "shadow":
        # Shadow mode
        shadow_config = {
            "model_type": model_type,
            "shadow_version": version_id,
            "production_version": current_version,
            "evaluation_period": 72,  # hours
            "success_criteria":
define_shadow_success_criteria(model_type)
        }

        start_shadow_mode(shadow_config)
        update_version_status(version_id, "shadow_mode")

    # Log deployment event
    log_deployment_event(
        model_type, version_id, current_version, deployment_strategy
    )

    return deployment_strategy

def rollback_model(model_type, problematic_version, reason):
    # Get version history
    version_history = get_version_history(model_type)

    # Find the previous stable version
    previous_stable = find_previous_stable_version(version_history,
problematic_version)

    if not previous_stable:
        raise Exception("No stable version found for rollback")

    # Execute rollback
    set_production_version(model_type, previous_stable)
    update_version_status(problematic_version, "rolled_back")
```

```
    update_version_status(previous_stable, "production")

    # Log rollback event
    rollback_metadata = {
        "model_type": model_type,
        "problematic_version": problematic_version,
        "rollback_version": previous_stable,
        "reason": reason,
        "timestamp": current_timestamp(),
        "executed_by": current_user()
    }

    log_rollback_event(rollback_metadata)

    # Trigger investigation
    trigger_rollback_investigation(problematic_version, reason)

    return previous_stable
```

9. Implementation Roadmap

A phased approach to implementing the recommendation engine, ensuring incremental delivery of value while managing complexity.

9.1 Phase 1: Foundation

Duration: 6-8 weeks

Objectives:
- Establish core data pipeline integration
- Implement basic feature engineering
- Develop initial profitability prediction models
- Create simple recommendation generation framework
- Set up monitoring and evaluation infrastructure

Key Deliverables:

1. **Data Integration**:
   - Connect to Vnish firmware API for miner telemetry
   - Integrate with Prohashing API for pool performance data
   - Set up market data collection from CoinGecko/CoinMarketCap
   - Implement data validation and cleaning processes

2. **Feature Store Setup**:
   - Configure Abacus.AI feature store
   - Define initial feature groups and schemas
   - Implement feature computation pipelines
   - Set up feature monitoring

3. **Basic Models**:
   - Develop coin profitability prediction model
   - Implement simple power optimization model
   - Create basic anomaly detection for hardware issues
   - Train and validate initial models

4. **Recommendation Framework**:
   - Design recommendation data structures
   - Implement basic recommendation generation logic
   - Create simple explanation generation
   - Develop recommendation storage and retrieval

5. **Web Application Integration**:
   - Design recommendation display components
   - Implement recommendation API endpoints
   - Create basic user feedback collection
   - Set up recommendation notification system

Success Criteria:
- End-to-end data flow from sources to recommendations
- Basic coin switching recommendations with >70% accuracy
- Simple power optimization recommendations
- Functional recommendation display in web application
- Initial monitoring dashboards operational

9.2 Phase 2: Core Recommendations

Duration: 8-10 weeks

Objectives:
- Enhance model accuracy and sophistication
- Expand recommendation types
- Implement personalization capabilities
- Develop comprehensive evaluation framework
- Establish continuous learning mechanisms

Key Deliverables:

1. **Advanced Feature Engineering**:
   - Implement cross-source feature derivation
   - Develop temporal feature extraction
   - Create market-specific feature transformations
   - Optimize feature selection for each model type

2. **Enhanced Models**:
   - Improve profitability prediction with LSTM models
   - Develop advanced power optimization using Bayesian methods
   - Implement comprehensive anomaly detection
   - Create initial hardware lifecycle models

3. **Expanded Recommendations**:
   - Implement merge mining optimization
   - Develop detailed power setting recommendations
   - Create maintenance scheduling recommendations
   - Implement basic hardware upgrade suggestions

4. **Personalization Framework**:
   - Develop user preference modeling
   - Implement recommendation filtering based on preferences
   - Create personalized explanation generation
   - Set up recommendation prioritization

5. **Evaluation System**:
   - Implement comprehensive metrics tracking
   - Develop A/B testing framework
   - Create detailed feedback analysis
   - Set up performance monitoring dashboards

Success Criteria:
- Profitability prediction accuracy >85%
- At least 4 recommendation types fully implemented
- Personalized recommendations based on user preferences
- Comprehensive evaluation metrics available
- Initial feedback loop operational

9.3 Phase 3: Advanced Features

Duration: 8-10 weeks

Objectives:
- Implement advanced ML techniques
- Develop sophisticated recommendation strategies
- Create comprehensive explanation system
- Enhance continuous learning capabilities
- Optimize system performance

Key Deliverables:

1. **Advanced ML Implementation**:
   - Deploy reinforcement learning for dynamic optimization
   - Implement ensemble methods for prediction robustness
   - Develop deep learning for complex pattern recognition
   - Create advanced time-series forecasting models

2. **Sophisticated Recommendations**:
   - Implement multi-objective optimization recommendations
   - Develop risk-aware recommendation strategies
   - Create long-term planning recommendations
   - Implement market-adaptive switching strategies

3. **Explanation System**:
   - Develop detailed explanation generation
   - Implement visualization of recommendation impacts
   - Create comparative analysis explanations
   - Develop confidence and uncertainty communication

4. **Enhanced Learning**:
   - Implement comprehensive drift detection
   - Develop automated model retraining
   - Create advanced feedback integration
   - Set up model versioning and lifecycle management

5. **Performance Optimization**:
   - Optimize inference latency for real-time recommendations
   - Implement efficient batch processing for scheduled recommendations
   - Develop caching strategies for frequent requests
   - Create resource-adaptive processing

Success Criteria:
- All recommendation types fully implemented
- Advanced explanation system operational
- Automated retraining based on feedback and drift

- Sub-second latency for real-time recommendations
- Comprehensive monitoring and alerting

9.4 Phase 4: Optimization and Scale

Duration: 6-8 weeks

Objectives:
- Fine-tune model performance
- Optimize system efficiency
- Enhance user experience
- Prepare for scale
- Implement advanced analytics

Key Deliverables:

1. **Model Fine-tuning**:
   - Optimize hyperparameters across all models
   - Implement advanced feature selection
   - Develop model-specific optimizations
   - Create specialized models for user segments

2. **System Efficiency**:
   - Optimize data pipeline for reduced latency
   - Implement efficient feature computation
   - Develop resource-aware scheduling
   - Create performance benchmarking framework

3. **User Experience Enhancements**:
   - Refine recommendation presentation
   - Implement advanced notification strategies
   - Develop user-specific dashboards
   - Create recommendation impact tracking

4. **Scalability Preparation**:
   - Implement horizontal scaling for inference
   - Develop efficient batch processing
   - Create load balancing strategies
   - Optimize resource utilization

5. **Advanced Analytics**:
   - Develop comprehensive performance analytics
   - Implement recommendation impact analysis

- Create user behavior analytics
- Develop system health analytics

Success Criteria:
- Model performance optimized across all types
- System capable of handling 10x current load
- Enhanced user experience with personalized dashboards
- Comprehensive analytics available for all aspects
- Full system documentation and operational procedures

10. Appendices

10.1 Feature Definitions

Detailed definitions of key features used in the recommendation engine.

Miner Telemetry Features:

Feature Name	Description	Source	Update Frequency	Used In
`hashrate_th_s`	Mining hashrate in TH/s	Vnish API	5 min	Profitability, Power Optimization
`power_consumption_w`	Power consumption in Watts	Vnish API	5 min	Power Optimization, Efficiency
`efficiency_j_th`	Energy efficiency in J/TH	Derived	5 min	Power Optimization, Profitability
`avg_chip_temp_c`	Average chip temperature in Celsius	Vnish API	5 min	Maintenance, Hardware Lifecycle
`max_chip_temp_c`	Maximum chip temperature in Celsius	Vnish API	5 min	Maintenance, Anomaly Detection
`fan_speed_percent`	Fan speed as percentage of maximum	Vnish API	5 min	Maintenance, Thermal Management
`accepted_shares`	Number of accepted shares	Vnish API	5 min	Performance, Anomaly Detection
`rejected_shares`	Number of rejected shares	Vnish API	5 min	Performance, Anomaly Detection
`hashrate_stability`	Standard deviation of hashrate over time	Derived	1 hour	Anomaly Detection, Maintenance
`power_efficiency_trend`	Trend in power efficiency over time	Derived	1 day	Hardware Lifecycle, Optimization
`thermal_efficiency`		Derived	1 hour	

Feature Name	Description	Source	Update Frequency	Used In
	Hashrate per degree of temperature			Thermal Management, Maintenance
`overclock_profile`	Current overclocking profile	Vnish API	15 min	Power Optimization, Performance
`uptime_hours`	Continuous operation time in hours	Derived	1 hour	Maintenance, Reliability
`error_rate`	Frequency of reported errors	Derived	1 hour	Anomaly Detection, Maintenance
`hashboard_variance`	Variance in performance across hashboards	Derived	1 hour	Maintenance, Hardware Lifecycle

Pool Performance Features:

Feature Name	Description	Source	Update Frequency	Used In
`effective_hashrate_th_s`	Effective hashrate as measured by pool	Prohashing API	15 min	Profitability, Performance
`reported_vs_effective_ratio`	Ratio of reported to effective hashrate	Derived	15 min	Performance, Anomaly Detection
`earnings_usd_24h`	Earnings in USD over past 24 hours	Prohashing API	1 hour	Profitability, Coin Selection
`earnings_per_th_usd`	Earnings per TH/s in USD	Derived	1 hour	Profitability, Coin Selection
`merge_mining_bonus_percent`	Additional earnings from merge mining as percentage	Derived	1 hour	Coin Selection, Profitability
`coin_profitability_rank`	Rank of coin profitability among all options	Derived	1 hour	Coin Selection, Switching
`pool_efficiency_score`	Efficiency score compared to other pools	Derived	6 hours	Pool Selection, Optimization
`difficulty_trend_7d`	Trend in mining difficulty over 7 days	Derived	6 hours	Profitability Prediction, Coin Selection
`reward_stability`	Stability of mining rewards over time	Derived	1 day	Risk Assessment, Coin Selection
		Derived	1 hour	

Feature Name	Description	Source	Update Frequency	Used In
`fee_adjusted_profitability`	Profitability adjusted for pool fees			Pool Selection, Profitability
`time_to_reward`	Average time between reward payments	Derived	1 day	Cash Flow, Pool Selection
`share_acceptance_rate`	Percentage of shares accepted by pool	Derived	1 hour	Performance, Network Optimization
`primary_coin_dominance`	Percentage of earnings from primary coin vs merge coins	Derived	1 day	Coin Selection, Strategy
`profitability_volatility`	Standard deviation of profitability over time	Derived	1 day	Risk Assessment, Strategy
`optimal_coin_switching_frequency`	Recommended frequency for coin switching	Derived	1 day	Switching Strategy, Optimization

Market Data Features:

Feature Name	Description	Source	Update Frequency	Used In
`price_usd`	Current price in USD	CoinGecko/CMC	15 min	Profitability, Coin Selection
`price_change_24h_percent`	24-hour price change percentage	CoinGecko/CMC	15 min	Trend Analysis, Coin Selection
`price_change_7d_percent`	7-day price change percentage	CoinGecko/CMC	15 min	Trend Analysis, Strategy
`market_cap_usd`	Market capitalization in USD	CoinGecko/CMC	15 min	Coin Selection, Risk Assessment
`volume_24h_usd`	24-hour trading volume in USD	CoinGecko/CMC	15 min	Liquidity, Risk Assessment
`volatility_30d`	30-day price volatility	Derived	1 day	Risk Assessment, Strategy
`network_hashrate`	Total network hashrate	CoinGecko/CMC	1 hour	Difficulty Prediction, Profitability
`network_difficulty`	Current mining difficulty	CoinGecko/CMC	1 hour	Profitability, Coin Selection
`block_reward_usd`	Block reward value in USD	Derived	1 hour	Profitability, Coin Selection
`transaction_fees_percent`	Transaction fees as percentage of total reward	Derived	1 hour	Profitability, Coin Selection
`halving_countdown_days`	Days until next reward halving	Derived	1 day	Strategy, Long-term Planning
`price_correlation_btc`	Price correlation with Bitcoin	Derived	1 day	Risk Assessment, Portfolio Strategy

Feature Name	Description	Source	Update Frequency	Used In
`market_cycle_position`	Position in market cycle (bull/bear/neutral)	Derived	1 day	Strategy, Timing
`mining_profitability_trend`	Trend in overall mining profitability	Derived	1 day	Strategy, Hardware Investment
`price_support_resistance_proximity`	Proximity to key support/resistance levels	Derived	1 day	Timing, Risk Assessment

Derived Cross-Source Features:

Feature Name	Description	Source	Update Frequency	Used In
`actual_profit_margin_percent`	Profit margin considering all costs	Derived	1 hour	Profitability, Strategy
`roi_current_hardware_days`	Estimated days to ROI for current hardware	Derived	1 day	Hardware Lifecycle, Investment
`optimal_power_limit_w`	Calculated optimal power limit	Derived	1 day	Power Optimization, Efficiency
`power_cost_per_dollar_earned`	Electricity cost per dollar of mining revenue	Derived	1 day	Efficiency, Profitability
`hardware_upgrade_benefit_score`	Potential benefit from hardware upgrade	Derived	1 week	Hardware Lifecycle, Investment
`maintenance_urgency_score`	Score indicating urgency of maintenance	Derived	1 day	Maintenance, Risk Management
`thermal_risk_score`	Risk score based on thermal conditions	Derived	1 hour	Maintenance, Risk Management
`efficiency_vs_market_percentile`	Efficiency percentile compared to market	Derived	1 week	Competitiveness, Strategy
`optimal_coin_allocation`	Recommended allocation across coins	Derived	1 day	Strategy, Profitability
`predicted_24h_profit_change`	Forecasted change in profit over next 24h	Derived	1 hour	Strategy, Coin Selection
		Derived	1 day	

Feature Name	Description	Source	Update Frequency	Used In
`hardware_failure_probability`	Probability of hardware failure			Maintenance, Risk Management
`opportunity_cost_current_strategy`	Opportunity cost of current mining strategy	Derived	1 day	Strategy, Optimization
`market_based_power_adjustment`	Recommended power adjustment based on market	Derived	1 hour	Power Optimization, Strategy
`optimal_firmware_settings`	Recommended firmware configuration	Derived	1 week	Optimization, Performance
`predicted_maintenance_benefit`	Expected benefit from recommended maintenance	Derived	1 day	Maintenance, ROI

10.2 Model Hyperparameters

Recommended hyperparameters for key models in the recommendation engine.

Profitability Prediction - XGBoost Model:

```
{
    "objective": "reg:squarederror",
    "learning_rate": 0.05,
    "max_depth": 6,
    "min_child_weight": 1,
    "subsample": 0.8,
    "colsample_bytree": 0.8,
    "gamma": 0,
    "alpha": 0.1,
    "lambda": 1,
    "n_estimators": 200,
    "early_stopping_rounds": 20,
    "seed": 42
}
```

Profitability Prediction - LSTM Model:

```
{
    "units": [128, 64, 32],
    "dropout": 0.2,
    "recurrent_dropout": 0.2,
    "activation": "relu",
    "recurrent_activation": "sigmoid",
    "optimizer": "adam",
    "learning_rate": 0.001,
    "batch_size": 32,
    "epochs": 100,
    "patience": 10,
    "sequence_length": 24,  # hours
    "features": [
        "price_usd", "price_change_24h_percent", "network_difficulty",
        "network_hashrate", "block_reward_usd", "transac-
tion_fees_percent",
        "market_cap_usd", "volume_24h_usd"
    ]
}
```

Power Optimization - Bayesian Optimization Model:

```
{
    "acquisition_function": "expected_improvement",
    "alpha": 0.0001,
    "n_initial_points": 10,
    "noise": "gaussian",
    "normalize_y": True,
    "kernel": "matern",
    "n_restarts_optimizer": 5,
    "random_state": 42,
    "search_bounds": {
        "power_limit": [0.7, 1.0],  # Percentage of rated power
        "frequency": [0.8, 1.1],    # Percentage of rated frequency
        "voltage": [0.9, 1.05]      # Percentage of rated voltage
    },
    "constraints": {
        "max_temperature": 75,       # Celsius
        "min_hashrate": 0.8,         # Percentage of rated hashrate
        "max_error_rate": 0.02       # Maximum acceptable error rate
    }
}
```

Anomaly Detection - Isolation Forest Model:

```
{
    "n_estimators": 100,
    "max_samples": "auto",
    "contamination": 0.05,
    "max_features": 1.0,
    "bootstrap": False,
    "n_jobs": -1,
    "random_state": 42,
    "warm_start": False,
    "features": [
        "hashrate_stability", "power_consumption_w", "efficiency_j_th",
        "avg_chip_temp_c", "max_chip_temp_c", "fan_speed_percent",
        "rejected_shares", "error_rate", "hashboard_variance",
        "reported_vs_effective_ratio"
    ]
}
```

Hardware Lifecycle - Survival Analysis Model:

```
{
    "model_type": "cox_ph",
    "alpha": 0.05,
    "ties": "efron",
    "penalizer": 0.1,
    "l1_ratio": 0.0,
    "robust": True,
    "features": [
        "uptime_hours", "avg_chip_temp_c", "thermal_efficiency",
        "power_efficiency_trend", "hashrate_stability", "error_rate",
        "hashboard_variance", "overclock_profile", "mainten-
ance_frequency"
    ],
    "duration_col": "time_to_failure",
    "event_col": "failure_observed"
}
```

Recommendation Ranking - LambdaMART Model:

```
{
    "objective": "lambdarank",
    "metric": "ndcg",
    "learning_rate": 0.05,
    "max_depth": 5,
    "min_child_weight": 1,
    "subsample": 0.8,
    "colsample_bytree": 0.8,
    "n_estimators": 100,
    "early_stopping_rounds": 10,
    "seed": 42,
    "features": [
        "predicted_impact", "confidence_score", "implementa-
tion_difficulty",
        "urgency_score", "user_preference_alignment", "historic-
al_acceptance"
    ]
}
```

10.3 Integration Code Samples

Example code snippets for integrating with Abacus.AI and implementing key components of the recommendation engine.

Feature Store Integration:

```
import abacusai
from datetime import datetime, timedelta

# Initialize Abacus.AI client
client = abacusai.ApiClient(api_key="YOUR_API_KEY")

# Define a feature group
def create_miner_telemetry_feature_group():
    feature_group = client.create_feature_group(
        name="miner_telemetry_features",
        description="Telemetry data from cryptocurrency miners",
        feature_definitions=[
            {"name": "miner_id", "dataType": "STRING", "tags": ["en-
tity_id"]},
            {"name": "timestamp", "dataType": "TIMESTAMP", "tags": ["ti
mestamp"]},
            {"name": "hashrate_th_s", "dataType": "FLOAT"},
            {"name": "power_consumption_w", "dataType": "FLOAT"},
            {"name": "efficiency_j_th", "dataType": "FLOAT"},
            {"name": "avg_chip_temp_c", "dataType": "FLOAT"},
            {"name": "max_chip_temp_c", "dataType": "FLOAT"},
            {"name": "fan_speed_percent", "dataType": "FLOAT"},
            {"name": "accepted_shares", "dataType": "INTEGER"},
            {"name": "rejected_shares", "dataType": "INTEGER"},
            {"name": "hashrate_stability", "dataType": "FLOAT"},
            {"name": "power_efficiency_trend", "dataType": "FLOAT"},
            {"name": "thermal_efficiency", "dataType": "FLOAT"},
            {"name": "overclock_profile", "dataType": "STRING"},
            {"name": "uptime_hours", "dataType": "FLOAT"},
            {"name": "error_rate", "dataType": "FLOAT"},
            {"name": "hashboard_variance", "dataType": "FLOAT"}
        ],
        primary_key="miner_id",
        timestamp_key="timestamp"
    )
    return feature_group

# Upload feature data
def upload_miner_telemetry_data(feature_group_id, miner_data):
    # Convert data to appropriate format
    formatted_data = []
    for record in miner_data:
        formatted_record = {
            "miner_id": record["miner_id"],
            "timestamp": datetime.fromtimestamp(record["timestamp"]).is
oformat(),
            "hashrate_th_s": record["hashrate"]["total"],
            "power_consumption_w": record["power"]["consumption"],
```

```
            "efficiency_j_th": record["power"]["efficiency"],
            "avg_chip_temp_c": record["temperature"]["avg_chip"],
            "max_chip_temp_c": record["temperature"]["max_chip"],
            "fan_speed_percent": record["fans"][0]["speed_percent"],
            "accepted_shares": record["shares"]["accepted"],
            "rejected_shares": record["shares"]["rejected"],
            # Derived features would be calculated here
            "hashrate_stability": calculate_hashrate_stability(record),
            "power_efficiency_trend": calcu-
late_efficiency_trend(record),
            "thermal_efficiency": calculate_thermal_efficiency(record),
            "overclock_profile": record["config"]["overclock_profile"],
            "uptime_hours": record["status"]["uptime"] / 3600,
            "error_rate": calculate_error_rate(record),
            "hashboard_variance": calculate_hashboard_variance(record)
        }
        formatted_data.append(formatted_record)

    # Upload to feature store
    upload_job = client.create_feature_group_upload_job(
        feature_group_id=feature_group_id,
        data=formatted_data
    )

    return upload_job.id

# Create a feature deployment for online serving
def deploy_feature_group(feature_group_id):
    deployment = client.create_feature_group_deployment(
        feature_group_id=feature_group_id,
        name="miner_telemetry_deployment",
        description="Deployment for real-time miner telemetry features"
    )
    return deployment.id

# Retrieve features for online inference
def get_online_features(deployment_id, miner_ids):
    features = client.get_deployed_feature_values(
        deployment_id=deployment_id,
        entity_ids=miner_ids
    )
    return features
```

Model Training Pipeline:

```
import abacusai
from datetime import datetime, timedelta

# Initialize Abacus.AI client
client = abacusai.ApiClient(api_key="YOUR_API_KEY")

# Create a training dataset
def create_profitability_training_dataset(feature_group_ids,
start_date, end_date):
    # Define point-in-time feature joining
    feature_joins = []
    for fg_id in feature_group_ids:
        feature_group = client.get_feature_group(fg_id)
        feature_joins.append({
            "featureGroupId": fg_id,
            "featureNames": [f["name"] for f in feature_group.fea-
ture_definitions],
            "lookbackWindow": "24h" if "market" in feature_group.name e
lse "1h"
        })

    # Create dataset
    dataset = client.create_feature_group_dataset(
        name="profitability_prediction_dataset",
        description="Dataset for training profitability prediction mod-
els",
        feature_group_id=feature_group_ids[0],  # Primary feature group
        feature_joins=feature_joins[1:],  # Additional feature groups
        start_date=start_date.isoformat(),
        end_date=end_date.isoformat(),
        target_feature="earnings_per_th_usd",  # Target to predict
        prediction_horizon="24h"  # Predict 24 hours ahead
    )

    return dataset.id

# Train a profitability prediction model
def train_profitability_model(dataset_id):
    # Create model version
    model_version = client.create_model_version(
        name="profitability_prediction_model",
        description="Model to predict mining profitability for
different coins",
        problem_type="REGRESSION",
        dataset_id=dataset_id,
        target="earnings_per_th_usd",
        training_config={
            "algorithm": "XGBOOST",
```

```
            "hyperparameters": {
                "objective": "reg:squarederror",
                "learning_rate": 0.05,
                "max_depth": 6,
                "min_child_weight": 1,
                "subsample": 0.8,
                "colsample_bytree": 0.8,
                "gamma": 0,
                "alpha": 0.1,
                "lambda": 1,
                "n_estimators": 200
            },
            "evaluation_metric": "RMSE",
            "validation_strategy": "TIME_SERIES_SPLIT",
            "validation_percent": 20
        }
    )

    # Start training
    training_job = client.create_training_job(
        model_version_id=model_version.id
    )

    return model_version.id, training_job.id

# Deploy a trained model
def deploy_model(model_version_id):
    deployment = client.create_deployment(
        model_version_id=model_version_id,
        name="profitability_prediction_deployment",
        description="Deployment for profitability prediction model",
        min_workers=1,
        max_workers=5,
        per_worker=10  # Requests per worker
    )

    return deployment.id

# Get predictions from deployed model
def predict_profitability(deployment_id, features):
    predictions = client.predict(
        deployment_id=deployment_id,
        data=features
    )

    return predictions
```

Recommendation Generation:

```
import abacusai
from datetime import datetime, timedelta
import json

# Initialize Abacus.AI client
client = abacusai.ApiClient(api_key="YOUR_API_KEY")

# Generate coin switching recommendations
def generate_coin_switching_recommendations(user_id, miner_ids):
    # Get user preferences
    user_preferences = get_user_preferences(user_id)

    # Get miner features
    miner_features = get_miner_features(miner_ids)

    # Get market data
    market_data = get_market_data()

    # Get pool performance data
    pool_data = get_pool_performance(user_id)

    # Prepare features for prediction
    prediction_features = prepare_prediction_features(
        miner_features, market_data, pool_data
    )

    # Get profitability predictions for different coins
    profitability_predictions = predict_coin_profitability(
        prediction_features
    )

    # Apply business rules and constraints
    filtered_predictions = apply_business_rules(
        profitability_predictions, user_preferences
    )

    # Generate recommendations
    recommendations = []
    for miner_id in miner_ids:
        current_coin = get_current_mining_coin(miner_id, pool_data)
        best_coin = get_best_coin(miner_id, filtered_predictions)

        # Only recommend switching if improvement is significant
        if should_recommend_switch(current_coin, best_coin,
user_preferences):
            recommendation = {
                "type": "coin_switching",
                "miner_id": miner_id,
```

```
                "current_coin": current_coin,
                "recommended_coin": best_coin["coin"],
                "expected_improvement":
best_coin["improvement_percent"],
                "confidence": best_coin["confidence"],
                "reasoning": generate_switching_reasoning(current_coin,
best_coin),
                "implementation_steps": gener-
ate_implementation_steps(miner_id, best_coin["coin"]),
                "timestamp": datetime.now().isoformat()
            }
            recommendations.append(recommendation)

    # Store recommendations
    store_recommendations(user_id, recommendations)

    return recommendations

# Generate power optimization recommendations
def generate_power_optimization_recommendations(user_id, miner_ids):
    # Get user preferences
    user_preferences = get_user_preferences(user_id)

    # Get miner features
    miner_features = get_miner_features(miner_ids)

    # Get market data
    market_data = get_market_data()

    # Get pool performance data
    pool_data = get_pool_performance(user_id)

    # Prepare features for prediction
    prediction_features = prepare_prediction_features(
        miner_features, market_data, pool_data
    )

    # Get optimal power settings predictions
    power_predictions = predict_optimal_power_settings(
        prediction_features
    )

    # Apply business rules and constraints
    filtered_predictions = apply_power_business_rules(
        power_predictions, user_preferences, miner_features
    )

    # Generate recommendations
    recommendations = []
```

```
    for miner_id in miner_ids:
        current_settings = get_current_power_settings(miner_id,
miner_features)
        optimal_settings = get_optimal_settings(miner_id,
filtered_predictions)

        # Only recommend changes if improvement is significant
        if should_recommend_power_change(current_settings, optim-
al_settings, user_preferences):
            recommendation = {
                "type": "power_optimization",
                "miner_id": miner_id,
                "current_power": current_settings["power_limit"],
                "recommended_power": optimal_settings["power_limit"],
                "expected_efficiency_improvement": optimal_settings["ef
ficiency_improvement"],
                "expected_hashrate_impact": optimal_settings["hashrate_
impact"],
                "net_profitability_impact": optimal_settings["profitab-
ility_impact"],
                "confidence": optimal_settings["confidence"],
                "reasoning": generate_power_reasoning(current_settings,
optimal_settings),
                "implementation_steps": gener-
ate_power_implementation_steps(miner_id, optimal_settings),
                "timestamp": datetime.now().isoformat()
            }
            recommendations.append(recommendation)

    # Store recommendations
    store_recommendations(user_id, recommendations)

    return recommendations

# Process user feedback
def process_recommendation_feedback(user_id, recommendation_id, feed-
back_type, details=None):
    # Get recommendation
    recommendation = get_recommendation(recommendation_id)

    # Record feedback
    feedback = {
        "user_id": user_id,
        "recommendation_id": recommendation_id,
        "recommendation_type": recommendation["type"],
        "feedback_type": feedback_type,
        "details": details,
        "timestamp": datetime.now().isoformat()
    }
```

```
    store_feedback(feedback)

    # Update recommendation status
    update_recommendation_status(recommendation_id, feedback_type)

    # Update user preferences based on feedback
    if feedback_type in ["accepted", "rejected"]:
        update_user_preferences(user_id, recommendation, feedback_type)

    # If implemented, schedule outcome verification
    if feedback_type == "implemented":
        schedule_outcome_verification(recommendation_id)

    # If rejected with reason, analyze for improvement
    if feedback_type == "rejected" and details:
        analyze_rejection_reason(recommendation, details)

    return feedback
```