Cryptocurrency Mining Monitoring System: Data Ingestion and Processing Pipeline Design

Table of Contents

1. Introduction

1.1 Purpose

This document outlines the design of a comprehensive data ingestion and processing pipeline for a cryptocurrency mining monitoring and optimization system. The pipeline is designed to collect, transform, and store data from multiple sources including ASIC miner firmware (Vnish), mining pools (Prohashing.com), and cryptocurrency market data APIs (CoinGecko, CoinMarket-Cap).

1.2 Scope

The data pipeline design encompasses:
- Data collection from multiple heterogeneous sources
- Data transformation and preprocessing workflows
- Schema definitions for structured data storage
- Polling frequency and scheduling strategies
- Integration methods with Abacus.AI's machine learning platform
- Data storage architecture and retention policies
- Error handling and monitoring mechanisms

1.3 System Overview

The cryptocurrency mining monitoring system aims to provide real-time insights and optimization recommendations for cryptocurrency mining operations, with a focus on merge mining strategies. The data pipeline serves as the foundation of this system, ensuring reliable, timely, and accurate data is available for analysis, visualization, and machine learning models.

2. Data Sources and Collection Methods

2.1 Vnish Firmware API

Vnish firmware provides detailed telemetry data from ASIC miners through its API endpoints.

Key Data Points:

- Hashrate metrics (overall and per hashboard)
- Temperature data (PCB, chip, and per hashboard)
- Power consumption and efficiency metrics
- Fan speed and cooling status
- Operational status and error codes
- Overclocking and performance settings

Primary Endpoints:

- `/cgi-bin/stats.cgi` - Real-time miner statistics
- `/cgi-bin/config.cgi` - Current configuration settings

Authentication:

- Basic HTTP authentication using miner credentials
- Secure credential storage in encrypted configuration

2.2 Prohashing.com API

Prohashing provides data about merge mining operations, profitability, and pool performance.

Key Data Points:

- Current and historical profitability metrics
- Merge mining configuration and performance
- Worker status and hashrate contribution
- Payout information and history
- Pool status and efficiency metrics

API Types:

- WAMP API for real-time data (primary)
- HTTP API for services that don’t support WAMP (secondary)

Authentication:

- API key-based authentication
- Secure key storage in encrypted configuration

2.3 Market Data APIs

External market data APIs provide cryptocurrency price, volume, and market capitalization information.

CoinGecko API:

- Comprehensive cryptocurrency market data
- Free tier with reasonable rate limits
- No API key required for basic usage

Key Endpoints:
- `/simple/price` - Current prices for specified cryptocurrencies

- `/coins/markets` - Market data including price, volume, and market cap
- `/coins/{id}/market_chart` - Historical price and volume data

CoinMarketCap API:

- Professional API with extensive market data
- Requires API key for all endpoints
- Free tier limited to 10,000 credits per month

Key Endpoints:
- `/cryptocurrency/listings/latest` - Latest market data sorted by market cap
- `/global-metrics/quotes/latest` - Global cryptocurrency market metrics

2.4 Collection Methods

Vnish Firmware Data Collection:

- Direct HTTP requests to each miner’s API endpoints
- Parallel collection from multiple miners
- Configurable retry logic with exponential backoff
- IP address and credential management for multiple miners

Prohashing Data Collection:

- WAMP client for real-time data subscription
- HTTP fallback for specific endpoints or when WAMP is unavailable
- Event-driven architecture for real-time updates
- Periodic polling for historical and aggregate data

Market Data Collection:

- HTTP requests to REST API endpoints
- Rate limiting compliance through request throttling
- Caching layer to minimize redundant requests
- Fallback between providers for reliability

3. Data Transformation and Preprocessing

3.1 Vnish Data Preprocessing

Normalization:

- Standardize units across different miner models (TH/s, GH/s, etc.)
- Convert temperature readings to consistent scale (Celsius)
- Normalize power metrics to standard units (Watts, J/TH)

Enrichment:

- Add miner metadata (model, firmware version, location)
- Calculate derived metrics (efficiency, uptime percentage)
- Flag anomalous readings based on historical patterns

Cleaning:

- Filter out invalid or corrupted readings
- Handle missing values through interpolation or flagging
- Remove duplicate entries from retry mechanisms

3.2 Prohashing Data Preprocessing

Normalization:

- Standardize profitability metrics to common units (USD/TH/day)
- Normalize timestamps to UTC for consistent time-series analysis
- Convert share counts to standardized difficulty units

Enrichment:

- Add pool metadata (algorithm, merge-mined coins)
- Calculate derived metrics (acceptance rate, effective hashrate)
- Correlate worker performance with miner telemetry

Aggregation:

- Summarize performance by time periods (hourly, daily)

- Group metrics by worker, algorithm, and coin
- Calculate moving averages for trend analysis

3.3 Market Data Preprocessing

Normalization:

- Standardize currency representations (USD, BTC)
- Normalize timestamps to UTC
- Reconcile coin identifiers across different APIs

Enrichment:

- Calculate price change percentages over custom time periods
- Derive volatility metrics
- Add mining-specific metrics (mining profitability, network difficulty)

Filtering:

- Focus on relevant coins for the mining operation
- Filter out low-quality or incomplete data points
- Prioritize data from more reliable sources when duplicates exist

3.4 Data Normalization and Enrichment

Cross-Source Integration:

- Join miner telemetry with pool performance data
- Correlate market movements with mining profitability
- Create unified time-series with consistent timestamps

Feature Engineering:

- Calculate efficiency metrics (cost per hash, revenue per watt)
- Derive profitability indicators
- Create technical indicators for market data
- Generate anomaly scores for operational metrics

Data Quality Assurance:

- Validate data consistency across sources
- Detect and flag outliers
- Implement data quality scoring
- Track lineage for auditability

4. Data Schemas

4.1 Miner Telemetry Schema

```
{
  "miner_telemetry": {
    "timestamp": "TIMESTAMP",
    "miner_id": "STRING",
    "ip_address": "STRING",
    "model": "STRING",
    "firmware_version": "STRING",
    "hashrate": {
      "total": "FLOAT",
      "unit": "STRING",
      "per_hashboard": [
        {
          "board_id": "INTEGER",
          "hashrate": "FLOAT",
          "status": "STRING"
        }
      ]
    },
    "temperature": {
      "ambient": "FLOAT",
      "avg_chip": "FLOAT",
      "max_chip": "FLOAT",
      "per_hashboard": [
        {
          "board_id": "INTEGER",
          "pcb_temp": "FLOAT",
          "chip_temp": "FLOAT"
        }
      ]
    },
    "power": {
      "consumption": "FLOAT",
      "efficiency": "FLOAT",
      "voltage": "FLOAT"
    },
    "fans": [
      {
        "fan_id": "INTEGER",
        "speed": "FLOAT",
        "speed_percent": "FLOAT",
        "status": "STRING"
      }
    ],
    "pool": {
      "url": "STRING",
      "user": "STRING",
      "status": "STRING"
    },
    "shares": {
      "accepted": "INTEGER",
      "rejected": "INTEGER",
      "stale": "INTEGER",
      "last_share_time": "TIMESTAMP"
    },
    "status": {
      "mining_status": "STRING",
      "uptime": "INTEGER",
      "errors": [
        {
          "code": "STRING",
          "message": "STRING",
          "severity": "STRING"
        }
      ]
    },
    "config": {
      "frequency": "FLOAT",
      "overclock_profile": "STRING",
      "power_limit": "FLOAT"
    }
  }
}
```

4.2 Mining Pool Schema

```
{
  "pool_performance": {
    "timestamp": "TIMESTAMP",
    "pool_id": "STRING",
    "worker_id": "STRING",
    "algorithm": "STRING",
    "hashrate": {
      "reported": "FLOAT",
      "effective": "FLOAT",
      "unit": "STRING"
    },
    "shares": {
      "accepted": "INTEGER",
      "rejected": "INTEGER",
      "stale": "INTEGER",
      "invalid": "INTEGER"
    },
    "earnings": {
      "amount": "FLOAT",
      "currency": "STRING",
      "time_period": "STRING"
    },
    "coins_mined": [
      {
        "coin_id": "STRING",
        "symbol": "STRING",
        "amount": "FLOAT",
        "usd_value": "FLOAT",
        "mining_type": "STRING"  // "primary" or "merge-mined"
      }
    ],
    "profitability": {
      "per_hash_rate": "FLOAT",
      "unit": "STRING",
      "time_period": "STRING"
    },
    "difficulty": "FLOAT",
    "status": "STRING"
  }
}
```

4.3 Market Data Schema

```
{
  "market_data": {
    "timestamp": "TIMESTAMP",
    "source": "STRING",
    "coin": {
      "id": "STRING",
      "symbol": "STRING",
      "name": "STRING"
    },
    "price": {
      "usd": "FLOAT",
      "btc": "FLOAT"
    },
    "market_cap": {
      "usd": "FLOAT",
      "btc": "FLOAT"
    },
    "volume_24h": {
      "usd": "FLOAT",
      "btc": "FLOAT"
    },
    "price_change": {
      "percent_1h": "FLOAT",
      "percent_24h": "FLOAT",
      "percent_7d": "FLOAT"
    },
    "mining_metrics": {
      "network_hashrate": "FLOAT",
      "network_difficulty": "FLOAT",
      "block_reward": "FLOAT",
      "block_time": "FLOAT",
      "mining_algorithm": "STRING"
    },
    "technical_indicators": {
      "rsi_14": "FLOAT",
      "ma_50": "FLOAT",
      "ma_200": "FLOAT",
      "volatility_30d": "FLOAT"
    }
  }
}
```

4.4 Derived Metrics Schema

```
{
  "derived_metrics": {
    "timestamp": "TIMESTAMP",
    "miner_id": "STRING",
    "time_period": "STRING",
    "operational_efficiency": {
      "hashrate_stability": "FLOAT",
      "power_efficiency": "FLOAT",
      "thermal_efficiency": "FLOAT",
      "uptime_percentage": "FLOAT"
    },
    "financial_metrics": {
      "revenue": "FLOAT",
      "cost": "FLOAT",
      "profit": "FLOAT",
      "roi": "FLOAT",
      "break_even_days": "FLOAT",
      "currency": "STRING"
    },
    "optimization_opportunities": [
      {
        "type": "STRING",
        "description": "STRING",
        "potential_impact": "FLOAT",
        "confidence": "FLOAT"
      }
    ],
    "anomaly_scores": {
      "hashrate": "FLOAT",
      "temperature": "FLOAT",
      "power": "FLOAT",
      "shares": "FLOAT",
      "overall": "FLOAT"
    },
    "forecasts": {
      "next_24h_hashrate": "FLOAT",
      "next_24h_revenue": "FLOAT",
      "next_7d_profitability": "FLOAT"
    }
  }
}
```

5. Polling Frequencies and Scheduling

5.1 Polling Strategy

Vnish Firmware API:

- **High-frequency metrics** (hashrate, temperature, power):
- Polling interval: 1-5 minutes
- Rationale: Critical operational metrics requiring near real-time monitoring
- **Configuration and status**:
- Polling interval: 15-30 minutes
- Rationale: Less frequent changes, lower priority for real-time updates
- **Adaptive polling**:
- Increase frequency during detected anomalies
- Decrease frequency during stable operation
- Implement backoff during API errors or rate limiting

Prohashing API:

- **WAMP real-time subscriptions**:
- Connection: Persistent WebSocket
- Updates: Event-driven (immediate upon changes)
- Heartbeat: 30 seconds to maintain connection
- **HTTP API fallback**:
- Worker status: 5-10 minutes
- Profitability data: 15-30 minutes
- Historical data: 1-6 hours
- **Payout and accounting data**:
- Polling interval: 1 hour
- Reconciliation process: Daily

Market Data APIs:

- **Price and basic market data**:
- Polling interval: 5-15 minutes
- Staggered requests across coins to avoid rate limits
- **Historical and technical data**:
- Polling interval: 1-6 hours
- Full refresh: Daily
- **Global market metrics**:
- Polling interval: 1 hour
- Comprehensive update: Daily

5.2 Scheduling Configuration

Time-based Scheduling:

- Implement cron-like scheduling for predictable, periodic tasks
- Configure timezone-aware scheduling to align with market hours
- Stagger related tasks to distribute load and avoid API rate limits

Event-based Scheduling:

- Trigger data collection based on system events (miner startup, configuration changes)
- Implement callback mechanisms for real-time data sources
- Chain dependent tasks to ensure data consistency

Priority-based Scheduling:

- Assign priority levels to different data collection tasks
- Implement preemptive scheduling for critical metrics
- Ensure high-priority tasks maintain their frequency during system load

5.3 Resource Optimization

Rate Limit Compliance:

- Track API usage across all data sources
- Implement token bucket algorithm for rate limiting
- Distribute requests evenly across time windows

Batch Processing:

- Group requests to the same API to minimize connection overhead
- Batch database operations for improved throughput
- Implement micro-batching for near-real-time processing

Resource Allocation:

- Allocate computing resources based on task priority
- Implement resource quotas for different data sources
- Scale resources dynamically based on workload

6. Integration with Abacus.AI

6.1 Data Ingestion Methods

REST API Integration:

- Use Abacus.AI REST API endpoints for data ingestion
- Implement authentication using API keys
- Handle rate limiting and retry logic

SDK Integration:

- Utilize Abacus.AI Python SDK for programmatic data ingestion
- Implement wrapper classes for different data sources
- Leverage SDK features for data validation and transformation

Batch Upload:

- Prepare data in compatible formats (CSV, JSON, Parquet)

- Use Abacus.AI batch upload endpoints for historical data
- Implement checkpointing for reliable uploads

6.2 Feature Store Integration

Feature Registration:

- Define feature groups aligned with data schemas
- Register features with appropriate metadata
- Implement versioning for feature definitions

Feature Computation:

- Define transformation logic for derived features
- Schedule regular feature computation jobs
- Implement on-demand feature computation for interactive analysis

Feature Serving:

- Configure online feature serving for real-time applications
- Implement feature retrieval for batch prediction
- Set up feature monitoring for drift detection

6.3 Real-time Processing

Stream Processing:

- Configure real-time data streams for critical metrics
- Implement windowed aggregations for streaming data
- Set up real-time anomaly detection

Event Processing:

- Define event patterns for operational alerts
- Implement complex event processing for multi-source correlation
- Configure event-driven workflows for automated responses

Real-time Dashboards:

- Push processed data to real-time visualization endpoints
- Implement websocket connections for dashboard updates
- Configure alert thresholds for real-time monitoring

6.4 Batch Processing

ETL Workflows:

- Define extraction, transformation, and loading workflows
- Schedule regular batch processing jobs
- Implement data quality checks in the workflow

Historical Analysis:

- Configure batch processing for historical data analysis
- Implement time-window aggregations for trend analysis
- Set up periodic model retraining based on historical data

Reporting:

- Generate scheduled reports from processed data
- Implement export functionality for external systems
- Configure notification mechanisms for report delivery

7. Data Storage and Retention Policies

7.1 Storage Architecture

Data Lake:

- Raw data storage for all collected data
- Organized by source, date, and data type
- Stored in optimized formats (Parquet, ORC)
- Supports schema evolution and versioning

Data Warehouse:

- Structured storage for processed and aggregated data
- Star schema design for analytical queries
- Optimized for complex queries and reporting
- Supports dimensional modeling for mining operations

Time-Series Database:

- Specialized storage for high-frequency telemetry data
- Optimized for time-based queries and aggregations
- Configurable downsampling for historical data
- Supports real-time dashboards and monitoring

Feature Store:

- Centralized repository for feature values
- Supports both online (low-latency) and offline (batch) access
- Maintains feature lineage and metadata
- Enables consistent feature serving across applications

7.2 Data Partitioning

Time-based Partitioning:

- Partition data by time periods (hourly, daily, monthly)
- Align partitioning with query patterns
- Implement partition pruning for query optimization

Entity-based Partitioning:

- Partition by logical entities (miner, pool, coin)
- Enable parallel processing across partitions
- Optimize for entity-specific queries

Hybrid Partitioning:

- Combine time and entity partitioning for complex datasets
- Implement dynamic partitioning based on data volume

- Configure partition management for optimal performance

7.3 Retention Policies

Raw Telemetry Data:

- High-resolution data (1-5 minute intervals): 7-30 days
- Medium-resolution data (hourly aggregates): 90-180 days
- Low-resolution data (daily aggregates): 1-3 years

Pool and Market Data:

- Detailed transaction data: 90 days
- Aggregated performance data: 1 year
- Historical profitability metrics: 2-3 years

Derived and Analytical Data:

- Operational metrics: 1 year
- Financial metrics: 3-5 years
- Benchmark and comparative data: Indefinite

System and Audit Logs:

- Error and warning logs: 30-90 days
- Audit trails: 1 year
- Configuration change history: 1 year

7.4 Data Archiving

Archiving Strategy:

- Move aged data to lower-cost storage tiers
- Implement data compression for archived data
- Maintain searchability through metadata indexing

Archive Access:

- Provide on-demand retrieval for archived data

- Implement batch restoration for analytical needs
- Maintain consistent query interface across active and archived data

Compliance and Governance:

- Align retention with regulatory requirements
- Implement secure deletion policies
- Maintain audit trails for data lifecycle events

8. Error Handling and Monitoring

8.1 Error Detection

Data Collection Errors:

- API connectivity failures
- Authentication and authorization errors
- Rate limiting and throttling issues
- Malformed responses and schema violations

Processing Errors:

- Transformation failures
- Data validation errors
- Resource exhaustion
- Timeout and performance issues

Integration Errors:

- Abacus.AI API connectivity issues
- Feature registration failures
- Data ingestion rejections
- Version compatibility problems

System Errors:

- Infrastructure failures

- Network connectivity issues
- Storage capacity problems
- Dependency failures

8.2 Error Recovery

Retry Mechanisms:

- Implement exponential backoff for transient errors
- Configure maximum retry attempts and timeouts
- Implement circuit breakers for persistent failures

Fallback Strategies:

- Define alternative data sources for critical metrics
- Implement degraded operation modes
- Utilize cached data during source unavailability

Data Reconciliation:

- Detect and fill data gaps after recovery
- Implement consistency checks across data sources
- Flag potentially affected derived metrics

Manual Intervention:

- Escalate unrecoverable errors to operators
- Provide diagnostic information for troubleshooting
- Implement manual override capabilities

8.3 Monitoring Framework

Operational Metrics:

- Data collection success rates
- API response times and error rates
- Processing latency and throughput
- Resource utilization (CPU, memory, network)

Data Quality Metrics:

- Completeness (missing values, coverage)
- Accuracy (validation failures, outliers)
- Consistency (cross-source validation)
- Timeliness (collection delays, processing backlog)

Business Metrics:

- Mining operation health indicators
- Profitability and efficiency metrics
- Optimization opportunity identification
- Anomaly detection effectiveness

System Health:

- Component availability and performance
- Integration status with external systems
- Resource utilization and scaling metrics
- Security and compliance indicators

8.4 Alerting System

Alert Levels:

- Critical: Immediate attention required, potential revenue impact
- Warning: Potential issues requiring investigation
- Informational: Notable events for awareness

Notification Channels:

- Email for non-urgent notifications
- SMS/mobile push for urgent alerts
- Integration with incident management systems
- Dashboard indicators for operational visibility

Alert Aggregation:

- Group related alerts to prevent alert fatigue

• Implement alert suppression during known issues

• Configure alert escalation for unacknowledged issues

Self-healing Actions:

• Automatic recovery procedures for known issues

• Scaling actions based on load metrics

• Temporary feature disablement during partial outages

9. Implementation Roadmap

9.1 Phase 1: Core Data Collection

Duration: 4-6 weeks

1. **Week 1-2: Infrastructure Setup**
   - Deploy base infrastructure components
   - Configure security and access controls
   - Establish development and testing environments

2. **Week 3-4: Basic Data Collection**
   - Implement Vnish API integration for critical metrics
   - Set up Prohashing HTTP API integration
   - Configure basic market data collection from CoinGecko

3. **Week 5-6: Data Storage and Basic Processing**
   - Implement data lake storage for raw data
   - Configure basic transformation pipelines
   - Set up initial Abacus.AI integration
   - Deploy monitoring for core components

9.2 Phase 2: Advanced Processing

Duration: 6-8 weeks

1. **Week 1-2: Enhanced Data Collection**
   - Implement WAMP integration for Prohashing real-time data
   - Expand market data collection to include technical indicators
   - Add support for multiple miner firmware versions

2. **Week 3-4: Advanced Transformation**
   - Implement cross-source data correlation
   - Develop feature engineering pipelines
   - Configure data quality validation framework

3. **Week 5-6: Analytical Storage**
   - Set up data warehouse for analytical queries
   - Implement time-series database for telemetry data
   - Configure data partitioning and lifecycle management

4. **Week 7-8: Abacus.AI Feature Store**
   - Define and register feature groups
   - Implement feature computation workflows
   - Configure online and offline feature serving

9.3 Phase 3: Optimization and Scaling

Duration: 4-6 weeks

1. **Week 1-2: Performance Optimization**
   - Optimize data collection for high-frequency sources
   - Implement caching strategies for API requests
   - Fine-tune database performance

2. **Week 3-4: Advanced Monitoring**
   - Implement comprehensive error handling
   - Deploy advanced monitoring dashboards
   - Configure alerting system with escalation paths

3. **Week 5-6: Scaling and Resilience**
   - Implement horizontal scaling for high-load components
   - Configure auto-recovery mechanisms
   - Conduct load testing and failure simulations
   - Finalize documentation and operational procedures

10. Appendices

10.1 API Reference

Vnish Firmware API

```
Base URL: http://<miner_IP_address>/
Authentication: Basic HTTP Authentication
Key Endpoints:
- /cgi-bin/stats.cgi (GET) - Real-time statistics
- /cgi-bin/config.cgi (GET/POST) - Configuration management
```

Prohashing API

```
WAMP API:
- URI: wss://prohashing.com:3333/ws
- Authentication: API Key

HTTP API:
- Base URL: https://prohashing.com/api/v1
- Authentication: API Key
- Key Endpoints:
  - /status - WAMP connection status
  - /profitability - Algorithm profitability metrics
```

CoinGecko API

```
Base URL: https://api.coingecko.com/api/v3
Authentication: None (free tier)
Rate Limits: 10-50 calls per minute
Key Endpoints:
- /simple/price - Current prices
- /coins/markets - Market data
- /coins/{id}/market_chart - Historical data
```

CoinMarketCap API

```
Base URL: https://pro-api.coinmarketcap.com/v1
Authentication: API Key required
Rate Limits: 10,000 credits per month (free tier)
Key Endpoints:
- /cryptocurrency/listings/latest - Latest market data
- /global-metrics/quotes/latest - Global metrics
```

10.2 Sample Code Snippets

Vnish API Data Collection

```python
import requests
import json
from requests.auth import HTTPBasicAuth
import time
import logging

logger = logging.getLogger(__name__)

class VnishCollector:
    def __init__(self, miner_ip, username, password):
        self.base_url = f"http://{miner_ip}"
        self.auth = HTTPBasicAuth(username, password)
        self.session = requests.Session()

    def get_stats(self):
        try:
            response = self.session.get(
                f"{self.base_url}/cgi-bin/stats.cgi",
                auth=self.auth,
                timeout=10
            )
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            logger.error(f"Error fetching stats from {self.base_url}:
{str(e)}")
            return None

    def get_config(self):
        try:
            response = self.session.get(
                f"{self.base_url}/cgi-bin/config.cgi",
                auth=self.auth,
                timeout=10
            )
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            logger.error(f"Error fetching config from {self.base_url}:
{str(e)}")
            return None

    def collect_telemetry(self):
        """Collect and format telemetry data from the miner"""
        stats = self.get_stats()
        config = self.get_config()

        if not stats or not config:
            return None

        # Extract and format the data according to the schema
        telemetry = {
            "timestamp": int(time.time()),
            "miner_id": stats["data"]["miner_id"],
            "ip_address": self.base_url.split("//")[1],
            "model": "ANTMINER_S19",  # This would be extracted from
stats or config
            "firmware_version": "Vnish 1.0",
# This would be extracted from stats or config
            "hashrate": {
                "total": stats["data"]["hashrate"],
                "unit": stats["data"]["hashrate_unit"],
                "per_hashboard": []
            },
            # Additional fields would be populated here
        }

        # Populate hashboard data
        for board_id, temp in stats["data"]["temperature"].items():
            if board_id.startswith("board"):
                board_num = int(board_id.replace("board", ""))
                telemetry["hashrate"]["per_hashboard"].append({
                    "board_id": board_num,
                    "hashrate": stats["data"]["hashrate"] / 3,  # Sim-
plified example
                    "status": "active"
                })

        # Additional processing would happen here

        return telemetry
```

Prohashing WAMP Client

```python
import asyncio
import json
from autobahn.asyncio.wamp import ApplicationSession, ApplicationRunner
import logging

logger = logging.getLogger(__name__)

class ProhashingWAMPClient(ApplicationSession):
    def __init__(self, config=None):
        ApplicationSession.__init__(self, config)
        self.api_key = config.extra["api_key"]
        self.callback = config.extra["callback"]
        self.subscriptions = []

    async def onJoin(self, details):
        logger.info("Session joined")

        # Subscribe to profitability updates
        try:
            sub = await self.subscribe(
                self.on_profitability_update,
                'com.prohashing.profitability'
            )
            self.subscriptions.append(sub)
            logger.info("Subscribed to profitability updates")
        except Exception as e:
            logger.error(f"Error subscribing to profitability updates:
{str(e)}")

        # Subscribe to worker status updates
        try:
            sub = await self.subscribe(
                self.on_worker_update,
                'com.prohashing.worker'
            )
            self.subscriptions.append(sub)
            logger.info("Subscribed to worker updates")
        except Exception as e:
            logger.error(f"Error subscribing to worker updates:
{str(e)}")

    def on_profitability_update(self, data):
        """Handle profitability update events"""
        try:
            # Process the profitability data
            processed_data = {
                "timestamp": int(time.time()),
                "source": "prohashing_wamp",
                "type": "profitability",
                "data": data
            }

            # Pass the data to the callback function
            self.callback(processed_data)
        except Exception as e:
            logger.error(f"Error processing profitability update: {str(
e)}")

    def on_worker_update(self, data):
        """Handle worker update events"""
        try:
            # Process the worker data
            processed_data = {
                "timestamp": int(time.time()),
                "source": "prohashing_wamp",
                "type": "worker",
                "data": data
            }

            # Pass the data to the callback function
            self.callback(processed_data)
        except Exception as e:
            logger.error(f"Error processing worker update: {str(e)}")

    def onDisconnect(self):
        logger.info("Session disconnected")

# Usage example
def data_callback(data):
    print(f"Received data: {json.dumps(data, indent=2)}")

runner = ApplicationRunner(
    url="wss://prohashing.com:3333/ws",
    realm="realm1",
    extra={"api_key": "your_api_key", "callback": data_callback}
)

# Run the client
loop = asyncio.get_event_loop()
runner.run(ProhashingWAMPClient, loop=loop)
```

CoinGecko Market Data Collection

```
import requests
import time
import logging
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

logger = logging.getLogger(__name__)

class CoinGeckoCollector:
    def __init__(self, coins=None, vs_currencies=None):
        self.base_url = "https://api.coingecko.com/api/v3"
        self.coins = coins or ["bitcoin", "ethereum", "litecoin", "doge
coin"]
        self.vs_currencies = vs_currencies or ["usd", "btc"]

        # Configure session with retry logic
        self.session = requests.Session()
        retries = Retry(
            total=5,
            backoff_factor=0.5,
            status_forcelist=[429, 500, 502, 503, 504]
        )
        self.session.mount('https://', HTTPAdapter(max_retries=re-
tries))

    def get_current_prices(self):
        """Get current prices for specified coins"""
        try:
            response = self.session.get(
                f"{self.base_url}/simple/price",
                params={
                    "ids": ",".join(self.coins),
                    "vs_currencies": ",".join(self.vs_currencies),
                    "include_market_cap": "true",
                    "include_24hr_vol": "true",
                    "include_24hr_change": "true"
                },
                timeout=10
            )
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            logger.error(f"Error fetching prices from CoinGecko: {str(e
)}")
            return None

    def get_market_data(self, limit=10):
        """Get detailed market data for specified coins"""
        try:
            response = self.session.get(
                f"{self.base_url}/coins/markets",
                params={
                    "vs_currency": "usd",
                    "ids": ",".join(self.coins),
                    "order": "market_cap_desc",
                    "per_page": limit,
                    "page": 1,
                    "sparkline": "false",
                    "price_change_percentage": "1h,24h,7d"
                },
                timeout=10
            )
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            logger.error(f"Error fetching market data from CoinGecko:
{str(e)}")
            return None

    def collect_market_data(self):
        """Collect and format market data according to schema"""
        market_data = self.get_market_data()

        if not market_data:
            return None

        formatted_data = []
        timestamp = int(time.time())

        for coin in market_data:
            data = {
                "timestamp": timestamp,
                "source": "coingecko",
                "coin": {
                    "id": coin["id"],
                    "symbol": coin["symbol"],
                    "name": coin["name"]
                },
                "price": {
                    "usd": coin["current_price"],
                    "btc": coin["current_price"] / market_data[0]["cur-
rent_price"] if coin["id"] != "bitcoin" else 1.0
                },
                "market_cap": {
                    "usd": coin["market_cap"]
                },
                "volume_24h": {
                    "usd": coin["total_volume"]
                },
                "price_change": {
                    "percent_1h": coin.get("price_change_percentage_1h_
in_currency", 0),
                    "percent_24h": coin["price_change_percentage_24h"],
                    "percent_7d": coin.get("price_change_percentage_7d_
in_currency", 0)
                }
                # Additional fields would be populated here
            }

            formatted_data.append(data)

        return formatted_data
```

10.3 Configuration Templates

Data Collection Configuration

```yaml
# data_collection_config.yaml

# General settings
general:
  environment: production
  log_level: INFO
  data_directory: /data/crypto_mining_monitor

# Vnish miners configuration
miners:
  - id: miner_01
    ip: 192.168.1.101
    username: admin
    password: ${MINER_PASSWORD}
    model: ANTMINER_S19
    location: rack_1_position_1
    polling_interval_seconds: 300

  - id: miner_02
    ip: 192.168.1.102
    username: admin
    password: ${MINER_PASSWORD}
    model: ANTMINER_S19
    location: rack_1_position_2
    polling_interval_seconds: 300

# Prohashing configuration
prohashing:
  api_key: ${PROHASHING_API_KEY}
  wamp:
    enabled: true
    url: wss://prohashing.com:3333/ws
    realm: realm1
    subscriptions:
      - com.prohashing.profitability
      - com.prohashing.worker
  http:
    enabled: true
    base_url: https://prohashing.com/api/v1
    polling_intervals:
      status: 300
      profitability: 900

# Market data configuration
market_data:
  coingecko:
    enabled: true
    coins:
      - bitcoin
      - ethereum
      - litecoin
      - dogecoin
    vs_currencies:
      - usd
      - btc
    polling_intervals:
      simple_price: 300
      market_data: 900
      historical: 3600

  coinmarketcap:
    enabled: true
    api_key: ${CMC_API_KEY}
    tier: free
    polling_intervals:
      listings: 900
      global_metrics: 3600
```

Abacus.AI Integration Configuration

```yaml
# abacus_integration_config.yaml

# General settings
general:
  project_id: crypto_mining_monitor
  api_key: ${ABACUS_API_KEY}
  environment: production

# Data ingestion configuration
data_ingestion:
  batch_size: 1000
  max_retries: 3
  retry_backoff_factor: 2
  timeout_seconds: 30

# Feature groups
feature_groups:
  - name: miner_telemetry
    description: "Telemetry data from cryptocurrency miners"
    entity_id_column: miner_id
    timestamp_column: timestamp
    features:
      - name: hashrate_th_s
        type: FLOAT
        description: "Mining hashrate in TH/s"
      - name: power_consumption_w
        type: FLOAT
        description: "Power consumption in Watts"
      - name: efficiency_j_th
        type: FLOAT
        description: "Energy efficiency in Joules per Terahash"
      - name: avg_temperature_c
        type: FLOAT
        description: "Average temperature in Celsius"

  - name: pool_performance
    description: "Mining pool performance metrics"
    entity_id_column: worker_id
    timestamp_column: timestamp
    features:
      - name: effective_hashrate
        type: FLOAT
        description: "Effective hashrate as measured by the pool"
      - name: accepted_shares
        type: INTEGER
        description: "Number of accepted shares"
      - name: earnings_usd
        type: FLOAT
```

```yaml
          description: "Earnings in USD"

    - name: market_data
      description: "Cryptocurrency market data"
      entity_id_column: coin.id
      timestamp_column: timestamp
      features:
        - name: price_usd
          type: FLOAT
          description: "Price in USD"
        - name: market_cap_usd
          type: FLOAT
          description: "Market capitalization in USD"
        - name: volume_24h_usd
          type: FLOAT
          description: "24-hour trading volume in USD"

# Real-time processing
real_time:
  enabled: true
  streaming_window_seconds: 300
  anomaly_detection:
    enabled: true
    sensitivity: medium
    alert_threshold: 0.8

# Batch processing
batch_processing:
  schedule: "0 */6 * * *"  # Every 6 hours
  lookback_days: 7
  feature_computation:
    enabled: true
    include_historical: true
```

Data Retention Configuration

```yaml
# data_retention_config.yaml

# General settings
general:
  timezone: UTC
  storage_base_path: /data/crypto_mining_monitor
  archive_base_path: /archive/crypto_mining_monitor

# Raw data retention
raw_data:
  miner_telemetry:
    high_resolution:
      retention_days: 30
      storage_format: parquet
      compression: snappy
      partition_by: [date, miner_id]

    medium_resolution:
      retention_days: 180
      aggregation_interval: hourly
      storage_format: parquet
      compression: snappy
      partition_by: [year, month, miner_id]

    low_resolution:
      retention_days: 1095  # 3 years
      aggregation_interval: daily
      storage_format: parquet
      compression: snappy
      partition_by: [year, miner_id]

  pool_data:
    transaction_data:
      retention_days: 90
      storage_format: parquet
      compression: snappy
      partition_by: [date, worker_id]

    performance_data:
      retention_days: 365
      storage_format: parquet
      compression: snappy
      partition_by: [year, month, worker_id]

  market_data:
    price_data:
      retention_days: 365
      storage_format: parquet
      compression: snappy
      partition_by: [year, month, coin_id]

    technical_data:
      retention_days: 730  # 2 years
      storage_format: parquet
      compression: snappy
      partition_by: [year, coin_id]

# Processed data retention
processed_data:
  operational_metrics:
    retention_days: 365
    storage_format: parquet
    compression: snappy
    partition_by: [year, month, metric_type]

  financial_metrics:
    retention_days: 1825  # 5 years
    storage_format: parquet
    compression: snappy
    partition_by: [year, metric_type]

  anomaly_detection:
    retention_days: 90
    storage_format: parquet
    compression: snappy
    partition_by: [date, anomaly_type]

# System data retention
system_data:
  logs:
    error_logs:
      retention_days: 90
      storage_format: text
      compression: gzip

    info_logs:
      retention_days: 30
      storage_format: text
      compression: gzip

  audit_trails:
    retention_days: 365
    storage_format: parquet
    compression: snappy
    partition_by: [year, month, event_type]

# Archiving configuration
```

```yaml
archiving:
  schedule: "0 0 * * 0"  # Weekly on Sunday at midnight
  parallel_jobs: 4
  verification_enabled: true
  notification_email: admin@example.com
```