Cryptocurrency Mining Fleet Monitoring API Integration Guide

Overview

This guide provides comprehensive technical documentation for integrating Vnish Hashcore Toolkit and Prohashing.com APIs into an intelligent cryptocurrency mining fleet monitoring application for 6-60 PPLNS scrypt mining machines.

System Architecture

- **Vnish Hashcore Toolkit**: Direct ASIC miner telemetry and control
- **Prohashing.com**: Mining pool performance and profitability monitoring
- **Fleet Scale**: 6-60 mining machines
- **Mining Type**: PPLNS scrypt algorithm focus

---

1. Vnish Hashcore Toolkit API

1.1 API Overview

The Vnish Hashcore Toolkit provides direct access to ASIC miner telemetry through embedded HTTP APIs on each mining device.

Key Characteristics:
- **Access Method**: HTTP REST API
- **Authentication**: IP-based (local network access)
- **Data Format**: JSON
- **Base URL**: `http://{miner_ip_address}/docs/`
- **Network Requirement**: Local network access to each miner

1.2 Available Endpoints

Core Telemetry Endpoints

```
GET http://{miner_ip}/summary
GET http://{miner_ip}/chips
GET http://{miner_ip}/chains
GET http://{miner_ip}/status
```

Real-time Telemetry Data

- **Hashrate**: Total and per-chip performance (TH/s, MH/s)
- **Temperature**: Chip, hashboard, water inlet/outlet sensors
- **Power Usage**: Real-time consumption in watts
- **Uptime**: Operational duration tracking
- **Chip Status**: Individual ASIC chip health and performance
- **Fan RPM**: Cooling system monitoring

1.3 JSON Data Format

Summary Response Structure

```
{
  "summary": {
    "hashrate": "150 TH/s",
    "temperature": {
      "chip": 75,
      "board": 65,
      "water_inlet": 20,
      "water_outlet": 25
    },
    "power": 3400,
    "uptime": 86400,
    "status": "mining"
  },
  "chips": [
    {
      "id": 1,
      "hashrate": "1.2 TH/s",
      "temperature": 75,
      "status": "active"
    },
    {
      "id": 2,
      "hashrate": "1.2 TH/s",
      "temperature": 74,
      "status": "active"
    }
  ],
  "chains": [
    {
      "id": 0,
      "chips": 126,
      "hashrate": "50 TH/s",
      "temperature": 75,
      "status": "healthy"
    }
  ]
}
```

1.4 Implementation Examples

Python Implementation

```python
import requests
import json
from typing import Dict, List, Optional

class VnishMinerAPI:
    def __init__(self, miner_ip: str, timeout: int = 10):
        self.miner_ip = miner_ip
        self.base_url = f"http://{miner_ip}"
        self.timeout = timeout

    def get_summary(self) -> Optional[Dict]:
        """Get miner summary data including hashrate, temperature, power"""
        try:
            response = requests.get(
                f"{self.base_url}/summary",
                timeout=self.timeout
            )
            response.raise_for_status()
            return response.json()
        except requests.RequestException as e:
            print(f"Error fetching summary from {self.miner_ip}: {e}")
            return None

    def get_chip_data(self) -> Optional[List[Dict]]:
        """Get individual chip performance data"""
        try:
            response = requests.get(
                f"{self.base_url}/chips",
                timeout=self.timeout
            )
            response.raise_for_status()
            return response.json()
        except requests.RequestException as e:
            print(f"Error fetching chip data from {self.miner_ip}: {e}")
            return None

    def get_telemetry(self) -> Dict:
        """Get comprehensive telemetry data"""
        summary = self.get_summary()
        chips = self.get_chip_data()

        return {
            "miner_ip": self.miner_ip,
            "timestamp": time.time(),
            "summary": summary,
            "chips": chips,
            "status": "online" if summary else "offline"
        }

# Fleet monitoring implementation
class VnishFleetMonitor:
    def __init__(self, miner_ips: List[str]):
        self.miners = [VnishMinerAPI(ip) for ip in miner_ips]

    def collect_fleet_data(self) -> List[Dict]:
        """Collect telemetry from all miners in fleet"""
        fleet_data = []
        for miner in self.miners:
            data = miner.get_telemetry()
            fleet_data.append(data)
        return fleet_data

    def get_fleet_summary(self) -> Dict:
        """Get aggregated fleet statistics"""
        fleet_data = self.collect_fleet_data()

        total_hashrate = 0
        total_power = 0
        online_miners = 0

        for miner_data in fleet_data:
            if miner_data["status"] == "online" and miner_data["summary"]:
                online_miners += 1
                # Parse hashrate (assuming format like "150 TH/s")
                hashrate_str = miner_data["summary"].get("hashrate", "0 TH/s")
                hashrate_val = float(hashrate_str.split()[0])
                total_hashrate += hashrate_val
                total_power += miner_data["summary"].get("power", 0)

        return {
            "total_miners": len(self.miners),
            "online_miners": online_miners,
            "total_hashrate_ths": total_hashrate,
            "total_power_watts": total_power,
            "efficiency_wth": total_power / total_hashrate if total_hashrate > 0 else 0
        }
```

cURL Examples

```bash
# Get miner summary
curl -X GET "http://192.168.1.100/summary" \
  -H "Accept: application/json"

# Get chip data
curl -X GET "http://192.168.1.100/chips" \
  -H "Accept: application/json"

# Health check
curl -X GET "http://192.168.1.100/status" \
  -H "Accept: application/json"
```

1.5 Integration Requirements

Network Configuration

- **Access Type**: Local network only
- **Port**: Standard HTTP (80) or HTTPS (443)
- **Firewall**: Ensure mining network accessibility
- **IP Discovery**: Network scanning for miner detection

Error Handling

```python
def robust_miner_request(miner_ip: str, endpoint: str, retries: int = 3) -> Option-
al[Dict]:
    """Robust request with retry logic and error handling"""
    for attempt in range(retries):
        try:
            response = requests.get(
                f"http://{miner_ip}/{endpoint}",
                timeout=10,
                headers={"Accept": "application/json"}
            )
            response.raise_for_status()
            return response.json()
        except requests.Timeout:
            print(f"Timeout on attempt {attempt + 1} for {miner_ip}")
        except requests.ConnectionError:
            print(f"Connection error on attempt {attempt + 1} for {miner_ip}")
        except requests.HTTPError as e:
            print(f"HTTP error {e.response.status_code} for {miner_ip}")
            break  # Don't retry on HTTP errors
        except json.JSONDecodeError:
            print(f"Invalid JSON response from {miner_ip}")

        if attempt < retries - 1:
            time.sleep(2 ** attempt)  # Exponential backoff

    return None
```

---

2. Prohashing.com API

2.1 API Overview

Prohashing provides both WAMP (real-time) and REST APIs for monitoring pool performance, worker status, and profitability metrics.

Key Characteristics:
- **Protocols**: WAMP (WebSocket) for real-time, HTTP for REST
- **Authentication**: API Key based
- **Data Format**: JSON
- **Rate Limits**: ~5,000 requests/hour (estimated)
- **Base URL**: Via WAMP proxy or direct WAMP connection

2.2 Available Endpoints

Core API Endpoints

```
GET /                       # Service health check
GET /status               # WAMP connection status
GET /profitability        # Algorithm profitability data
```

PPLNS Scrypt Mining Data

- **Algorithm ID**: 1 (Scrypt)

- **Worker Status**: Real-time connection monitoring
- **Earnings Data**: USD/BTC profitability metrics
- **Pool Performance**: Server status and update timestamps

2.3 Authentication Setup

Environment Configuration

```
export PROHASHING_API_KEY="your_api_key_here"
export PROHASHING_SERVER_HOST="localhost"
export PROHASHING_SERVER_PORT="3000"
```

API Key Management

```
import os
from typing import Optional

class ProhashingAuth:
    def __init__(self):
        self.api_key = os.getenv('PROHASHING_API_KEY')
        if not self.api_key:
            raise ValueError("PROHASHING_API_KEY environment variable required")

    def get_headers(self) -> Dict[str, str]:
        return {
            "Authorization": f"Bearer {self.api_key}",
            "Content-Type": "application/json",
            "Accept": "application/json"
        }
```

2.4 Data Formats

Status Response

```
{
  "connected": true,
  "updating": true,
  "lastUpdates": {
    "profitability": "2019-01-15T00:00:52+00:00"
  }
}
```

Profitability Response (Scrypt Focus)

```
{
  "1": {
    "algorithm_name": "Scrypt",
    "usd": 0.0025041009705167,
    "btc": 6.8628068694276e-7,
    "max_usd": 0.0052676028280028,
    "max_btc": 1.4445957481592e-6,
    "percentile_usd": 0.0023765266674273,
    "percentile_btc": 6.5173886514719e-7,
    "data_timestamp": 1547510680.4959,
    "server_timestamp": 1547510683.3438,
    "server_id": 4
  }
}
```

2.5 Implementation Examples

Python Implementation

```python
import requests
import time
from datetime import datetime
from typing import Dict, Optional

class ProhashingAPI:
    def __init__(self, api_key: str, base_url: str = "http://localhost:3000"):
        self.api_key = api_key
        self.base_url = base_url
        self.session = requests.Session()
        self.session.headers.update({
            "Authorization": f"Bearer {api_key}",
            "Content-Type": "application/json"
        })

    def health_check(self) -> bool:
        """Check if API service is available"""
        try:
            response = self.session.get(f"{self.base_url}/", timeout=10)
            return response.status_code == 200
        except requests.RequestException:
            return False

    def get_status(self) -> Optional[Dict]:
        """Get WAMP connection status and last update times"""
        try:
            response = self.session.get(f"{self.base_url}/status", timeout=10)
            response.raise_for_status()
            return response.json()
        except requests.RequestException as e:
            print(f"Error fetching status: {e}")
            return None

    def get_profitability(self) -> Optional[Dict]:
        """Get algorithm profitability data"""
        try:
            response = self.session.get(f"{self.base_url}/profitability", timeout=10)
            response.raise_for_status()
            return response.json()
        except requests.RequestException as e:
            print(f"Error fetching profitability: {e}")
            return None

    def get_scrypt_profitability(self) -> Optional[Dict]:
        """Get Scrypt-specific profitability data"""
        profitability = self.get_profitability()
        if profitability and "1" in profitability:
            return profitability["1"]  # Algorithm ID 1 = Scrypt
        return None

    def is_data_fresh(self, max_age_minutes: int = 10) -> bool:
        """Check if profitability data is fresh"""
        status = self.get_status()
        if not status or not status.get("updating"):
            return False

        last_update = status.get("lastUpdates", {}).get("profitability")
        if not last_update:
            return False

        try:
            update_time = datetime.fromisoformat(last_update.replace('Z', '+00:00'))
            age_minutes = (datetime.now(update_time.tzinfo) - update_time).total_second
s() / 60
            return age_minutes <= max_age_minutes
        except (ValueError, TypeError):
            return False

# Worker monitoring implementation
class ProhashingWorkerMonitor:
    def __init__(self, api_key: str):
        self.api = ProhashingAPI(api_key)
        self.last_profitability = None
        self.last_update = None

    def monitor_scrypt_performance(self) -> Dict:
        """Monitor Scrypt mining performance and profitability"""
        status = self.api.get_status()
        scrypt_data = self.api.get_scrypt_profitability()

        performance_data = {
            "timestamp": time.time(),
            "api_connected": status.get("connected", False) if status else False,
            "data_updating": status.get("updating", False) if status else False,
            "data_fresh": self.api.is_data_fresh(),
            "scrypt_profitability": scrypt_data
        }

        if scrypt_data:
            performance_data.update({
                "usd_per_hash": scrypt_data.get("usd", 0),
                "btc_per_hash": scrypt_data.get("btc", 0),
                "max_usd_observed": scrypt_data.get("max_usd", 0),
                "percentile_usd": scrypt_data.get("percentile_usd", 0)
            })

        return performance_data
```

Worker Configuration

```
def configure_scrypt_worker(worker_name: str, pool_address: str) -> str:
    """Generate proper worker configuration for Scrypt mining"""
    # Scrypt workers should specify algorithm in password
    return f"stratum+tcp://{worker_name}:a=scrypt@{pool_address}"

# Example configuration
worker_config = configure_scrypt_worker(
    worker_name="miner001",
    pool_address="prohashing.com:3333"
)
```

2.6 Rate Limiting and Error Handling

Rate Limit Management

```python
import time
from functools import wraps

class RateLimiter:
    def __init__(self, max_requests: int = 5000, window_hours: int = 1):
        self.max_requests = max_requests
        self.window_seconds = window_hours * 3600
        self.requests = []

    def can_make_request(self) -> bool:
        now = time.time()
        # Remove old requests outside the window
        self.requests = [req_time for req_time in self.requests
                        if now - req_time < self.window_seconds]
        return len(self.requests) < self.max_requests

    def record_request(self):
        self.requests.append(time.time())

def rate_limited(rate_limiter: RateLimiter):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if not rate_limiter.can_make_request():
                raise Exception("Rate limit exceeded")

            result = func(*args, **kwargs)
            rate_limiter.record_request()
            return result
        return wrapper
    return decorator

# Usage
prohashing_limiter = RateLimiter(max_requests=4500)  # Leave buffer

@rate_limited(prohashing_limiter)
def safe_api_call(api_method, *args, **kwargs):
    return api_method(*args, **kwargs)
```

3. Data Synchronization Patterns

3.1 Polling Strategy

Vnish Miner Polling

```python
import asyncio
import aiohttp
from typing import List, Dict

class AsyncVnishMonitor:
    def __init__(self, miner_ips: List[str], poll_interval: int = 30):
        self.miner_ips = miner_ips
        self.poll_interval = poll_interval
        self.running = False

    async def poll_miner(self, session: aiohttp.ClientSession, miner_ip: str) -> Dict:
        """Poll single miner asynchronously"""
        try:
            async with session.get(
                f"http://{miner_ip}/summary",
                timeout=aiohttp.ClientTimeout(total=10)
            ) as response:
                if response.status == 200:
                    data = await response.json()
                    return {
                        "miner_ip": miner_ip,
                        "timestamp": time.time(),
                        "status": "online",
                        "data": data
                    }
        except Exception as e:
            return {
                "miner_ip": miner_ip,
                "timestamp": time.time(),
                "status": "offline",
                "error": str(e)
            }

    async def poll_all_miners(self) -> List[Dict]:
        """Poll all miners concurrently"""
        async with aiohttp.ClientSession() as session:
            tasks = [self.poll_miner(session, ip) for ip in self.miner_ips]
            return await asyncio.gather(*tasks)

    async def continuous_monitoring(self, callback=None):
        """Continuous monitoring loop"""
        self.running = True
        while self.running:
            try:
                results = await self.poll_all_miners()
                if callback:
                    await callback(results)
                await asyncio.sleep(self.poll_interval)
            except Exception as e:
                print(f"Monitoring error: {e}")
                await asyncio.sleep(5)  # Brief pause on error
```

Prohashing Data Synchronization

```python
class ProhashingDataSync:
    def __init__(self, api_key: str, sync_interval: int = 60):
        self.api = ProhashingAPI(api_key)
        self.sync_interval = sync_interval
        self.last_sync = None
        self.cached_data = {}

    async def sync_profitability_data(self) -> Dict:
        """Sync profitability data with caching"""
        try:
            # Check if data is fresh enough
            if (self.last_sync and
                time.time() - self.last_sync < self.sync_interval and
                self.api.is_data_fresh()):
                return self.cached_data

            # Fetch fresh data
            profitability = self.api.get_profitability()
            status = self.api.get_status()

            if profitability:
                self.cached_data = {
                    "timestamp": time.time(),
                    "profitability": profitability,
                    "status": status,
                    "scrypt_data": profitability.get("1", {})
                }
                self.last_sync = time.time()

            return self.cached_data

        except Exception as e:
            print(f"Sync error: {e}")
            return self.cached_data  # Return cached data on error
```

3.2 Real-time Integration Pattern

Combined Monitoring System

```python
class FleetMonitoringSystem:
    def __init__(self, miner_ips: List[str], prohashing_api_key: str):
        self.vnish_monitor = AsyncVnishMonitor(miner_ips, poll_interval=30)
        self.prohashing_sync = ProhashingDataSync(prohashing_api_key, sync_interval=60)
        self.data_store = {}

    async def process_fleet_data(self, miner_data: List[Dict]):
        """Process and correlate fleet data"""
        # Get current profitability data
        pool_data = await self.prohashing_sync.sync_profitability_data()

        # Calculate fleet metrics
        fleet_metrics = self.calculate_fleet_metrics(miner_data, pool_data)

        # Store data
        self.data_store[time.time()] = {
            "miners": miner_data,
            "pool": pool_data,
            "metrics": fleet_metrics
        }

        # Trigger alerts if needed
        await self.check_alerts(fleet_metrics)

    def calculate_fleet_metrics(self, miner_data: List[Dict], pool_data: Dict) -> Dict:
        """Calculate comprehensive fleet metrics"""
        online_miners = [m for m in miner_data if m["status"] == "online"]

        total_hashrate = 0
        total_power = 0
        avg_temp = 0

        for miner in online_miners:
            if miner.get("data"):
                # Parse hashrate
                hashrate_str = miner["data"].get("hashrate", "0 TH/s")
                hashrate_val = float(hashrate_str.split()[0])
                total_hashrate += hashrate_val

                # Sum power
                total_power += miner["data"].get("power", 0)

                # Average temperature
                temp_data = miner["data"].get("temperature", {})
                if isinstance(temp_data, dict):
                    avg_temp += temp_data.get("chip", 0)
                else:
                    avg_temp += temp_data

        if online_miners:
            avg_temp /= len(online_miners)

        # Calculate profitability
        scrypt_profitability = pool_data.get("scrypt_data", {}).get("usd", 0)
        estimated_daily_usd = total_hashrate * scrypt_profitability * 24 * 3600

        return {
            "total_miners": len(miner_data),
            "online_miners": len(online_miners),
            "total_hashrate_ths": total_hashrate,
            "total_power_watts": total_power,
            "average_temperature": avg_temp,
            "efficiency_wth": total_power / total_hashrate if total_hashrate > 0 else
0,
            "estimated_daily_usd": estimated_daily_usd,
            "scrypt_profitability_usd": scrypt_profitability
        }

    async def start_monitoring(self):
        """Start the complete monitoring system"""
        await self.vnish_monitor.continuous_monitoring(
            callback=self.process_fleet_data
        )
```

4. Best Practices for Fleet-Scale Monitoring

4.1 Error Handling and Resilience

Circuit Breaker Pattern

```
import time
from enum import Enum

class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitBreaker:
    def __init__(self, failure_threshold: int = 5, timeout: int = 60):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.failure_count = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    def call(self, func, *args, **kwargs):
        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time > self.timeout:
                self.state = CircuitState.HALF_OPEN
            else:
                raise Exception("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)
            self.on_success()
            return result
        except Exception as e:
            self.on_failure()
            raise e

    def on_success(self):
        self.failure_count = 0
        self.state = CircuitState.CLOSED

    def on_failure(self):
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.failure_threshold:
            self.state = CircuitState.OPEN

# Usage with APIs
vnish_breaker = CircuitBreaker(failure_threshold=3, timeout=30)
prohashing_breaker = CircuitBreaker(failure_threshold=5, timeout=60)

def safe_vnish_call(miner_api, method_name, *args, **kwargs):
    method = getattr(miner_api, method_name)
    return vnish_breaker.call(method, *args, **kwargs)
```

4.2 Data Storage and Persistence

Time-Series Data Storage

```
import sqlite3
import json
from datetime import datetime, timedelta

class FleetDataStore:
    def __init__(self, db_path: str = "fleet_monitoring.db"):
        self.db_path = db_path
        self.init_database()

    def init_database(self):
        """Initialize database schema"""
        with sqlite3.connect(self.db_path) as conn:
            conn.execute("""
                CREATE TABLE IF NOT EXISTS miner_telemetry (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    timestamp REAL NOT NULL,
                    miner_ip TEXT NOT NULL,
                    hashrate REAL,
                    temperature REAL,
                    power REAL,
                    status TEXT,
                    raw_data TEXT
                )
            """)

            conn.execute("""
                CREATE TABLE IF NOT EXISTS pool_data (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    timestamp REAL NOT NULL,
                    algorithm TEXT,
                    usd_profitability REAL,
                    btc_profitability REAL,
                    raw_data TEXT
                )
            """)

            conn.execute("""
                CREATE INDEX IF NOT EXISTS idx_miner_timestamp
                ON miner_telemetry(miner_ip, timestamp)
            """)

    def store_miner_data(self, miner_data: List[Dict]):
        """Store miner telemetry data"""
        with sqlite3.connect(self.db_path) as conn:
            for data in miner_data:
                if data["status"] == "online" and data.get("data"):
                    telemetry = data["data"]
                    hashrate_str = telemetry.get("hashrate", "0 TH/s")
                    hashrate = float(hashrate_str.split()[0])

                    temp_data = telemetry.get("temperature", {})
                    temperature = temp_data.get("chip", 0) if isinstance(temp_data, dic
t) else temp_data

                    conn.execute("""
                        INSERT INTO miner_telemetry
                        (timestamp, miner_ip, hashrate, temperature, power, status,
raw_data)
```

```
                    VALUES (?, ?, ?, ?, ?, ?, ?)
                """, (
                    data["timestamp"],
                    data["miner_ip"],
                    hashrate,
                    temperature,
                    telemetry.get("power", 0),
                    data["status"],
                    json.dumps(telemetry)
                ))

    def store_pool_data(self, pool_data: Dict):
        """Store pool profitability data"""
        if not pool_data.get("scrypt_data"):
            return

        scrypt_data = pool_data["scrypt_data"]
        with sqlite3.connect(self.db_path) as conn:
            conn.execute("""
                INSERT INTO pool_data
                (timestamp, algorithm, usd_profitability, btc_profitability, raw_data)
                VALUES (?, ?, ?, ?, ?)
            """, (
                pool_data["timestamp"],
                "Scrypt",
                scrypt_data.get("usd", 0),
                scrypt_data.get("btc", 0),
                json.dumps(scrypt_data)
            ))

    def get_fleet_history(self, hours: int = 24) -> Dict:
        """Get fleet performance history"""
        cutoff_time = time.time() - (hours * 3600)

        with sqlite3.connect(self.db_path) as conn:
            # Get miner data
            miner_cursor = conn.execute("""
                SELECT timestamp, miner_ip, hashrate, temperature, power, status
                FROM miner_telemetry
                WHERE timestamp > ?
                ORDER BY timestamp
            """, (cutoff_time,))

            miner_history = miner_cursor.fetchall()

            # Get pool data
            pool_cursor = conn.execute("""
                SELECT timestamp, usd_profitability, btc_profitability
                FROM pool_data
                WHERE timestamp > ?
                ORDER BY timestamp
            """, (cutoff_time,))

            pool_history = pool_cursor.fetchall()

        return {
            "miner_history": miner_history,
            "pool_history": pool_history
        }
```

4.3 Alerting and Monitoring

Alert System

```python
from dataclasses import dataclass
from typing import Callable, List
from enum import Enum

class AlertSeverity(Enum):
    INFO = "info"
    WARNING = "warning"
    CRITICAL = "critical"

@dataclass
class Alert:
    severity: AlertSeverity
    message: str
    miner_ip: str = None
    metric_value: float = None
    threshold: float = None
    timestamp: float = None

class FleetAlertSystem:
    def __init__(self):
        self.alert_handlers: List[Callable[[Alert], None]] = []
        self.thresholds = {
            "temperature_warning": 80,
            "temperature_critical": 90,
            "hashrate_drop_percent": 20,
            "power_spike_percent": 30,
            "offline_duration_minutes": 5
        }

    def add_alert_handler(self, handler: Callable[[Alert], None]):
        """Add alert handler (email, slack, etc.)"""
        self.alert_handlers.append(handler)

    def check_miner_alerts(self, current_data: List[Dict], historical_data: Dict =
None):
        """Check for miner-specific alerts"""
        for miner in current_data:
            miner_ip = miner["miner_ip"]

            # Check offline status
            if miner["status"] == "offline":
                alert = Alert(
                    severity=AlertSeverity.CRITICAL,
                    message=f"Miner {miner_ip} is offline",
                    miner_ip=miner_ip,
                    timestamp=time.time()
                )
                self._trigger_alert(alert)
                continue

            if not miner.get("data"):
                continue

            telemetry = miner["data"]

            # Temperature alerts
            temp_data = telemetry.get("temperature", {})
            chip_temp = temp_data.get("chip", 0) if isinstance(temp_data, dict) else
```

```
temp_data

            if chip_temp > self.thresholds["temperature_critical"]:
                alert = Alert(
                    severity=AlertSeverity.CRITICAL,
                    message=f"Critical temperature on {miner_ip}",
                    miner_ip=miner_ip,
                    metric_value=chip_temp,
                    threshold=self.thresholds["temperature_critical"],
                    timestamp=time.time()
                )
                self._trigger_alert(alert)
            elif chip_temp > self.thresholds["temperature_warning"]:
                alert = Alert(
                    severity=AlertSeverity.WARNING,
                    message=f"High temperature on {miner_ip}",
                    miner_ip=miner_ip,
                    metric_value=chip_temp,
                    threshold=self.thresholds["temperature_warning"],
                    timestamp=time.time()
                )
                self._trigger_alert(alert)

            # Hashrate drop detection (requires historical data)
            if historical_data:
                self._check_hashrate_drop(miner_ip, telemetry, historical_data)

    def _check_hashrate_drop(self, miner_ip: str, current_telemetry: Dict, historic-
al_data: Dict):
        """Check for significant hashrate drops"""
        current_hashrate_str = current_telemetry.get("hashrate", "0 TH/s")
        current_hashrate = float(current_hashrate_str.split()[0])

        # Get average hashrate from last hour
        recent_data = [
            row for row in historical_data.get("miner_history", [])
            if row[1] == miner_ip and time.time() - row[0] < 3600
        ]

        if recent_data:
            avg_hashrate = sum(row[2] for row in recent_data) / len(recent_data)
            drop_percent = ((avg_hashrate - current_hashrate) / avg_hashrate) * 100

            if drop_percent > self.thresholds["hashrate_drop_percent"]:
                alert = Alert(
                    severity=AlertSeverity.WARNING,
                    message=f"Hashrate drop of {drop_percent:.1f}% on {miner_ip}",
                    miner_ip=miner_ip,
                    metric_value=current_hashrate,
                    threshold=avg_hashrate,
                    timestamp=time.time()
                )
                self._trigger_alert(alert)

    def _trigger_alert(self, alert: Alert):
        """Trigger all registered alert handlers"""
        for handler in self.alert_handlers:
            try:
                handler(alert)
```

```python
            except Exception as e:
                print(f"Alert handler error: {e}")

# Example alert handlers
def email_alert_handler(alert: Alert):
    """Send email alert (implement with your email service)"""
    print(f"EMAIL ALERT: {alert.severity.value.upper()} - {alert.message}")

def slack_alert_handler(alert: Alert):
    """Send Slack alert (implement with Slack API)"""
    print(f"SLACK ALERT: {alert.severity.value.upper()} - {alert.message}")

# Usage
alert_system = FleetAlertSystem()
alert_system.add_alert_handler(email_alert_handler)
alert_system.add_alert_handler(slack_alert_handler)
```

---

5. Implementation Recommendations

5.1 System Architecture

Recommended Technology Stack

- **Backend**: Python 3.8+ with asyncio for concurrent operations
- **Database**: SQLite for development, PostgreSQL for production
- **Monitoring**: Prometheus + Grafana for metrics visualization
- **Alerting**: Custom alert system with email/Slack integration
- **API Framework**: FastAPI for REST API endpoints
- **Task Queue**: Celery with Redis for background tasks

Deployment Architecture

```
┌─────────────────┐    ┌───────────────────┐    ┌─────────────────┐
│   Web Dashboard │    │  API Gateway      │    │  Monitoring     │
│   (React/Vue)   │◄──►│  (FastAPI)        │◄──►│  Service        │
└─────────────────┘    └───────────────────┘    └─────────────────┘
                                │                        │
                                ▼                        ▼
                       ┌───────────────────┐    ┌─────────────────┐
                       │   Database        │    │  Alert System   │
                       │  (PostgreSQL)     │    │  (Email/Slack)  │
                       └───────────────────┘    └─────────────────┘
                                │
                                ▼
                       ┌───────────────────┐
                       │  Mining Network   │
                       │  (6-60 Miners)    │
                       └───────────────────┘
```

5.2 Configuration Management

Configuration File Structure

```yaml
# config.yaml
fleet:
  miner_ips:
    - "192.168.1.100"
    - "192.168.1.101"
    - "192.168.1.102"
  poll_interval_seconds: 30
  timeout_seconds: 10

prohashing:
  api_key: "${PROHASHING_API_KEY}"
  base_url: "http://localhost:3000"
  sync_interval_seconds: 60
  rate_limit_requests_per_hour: 4500

database:
  type: "postgresql"  # or "sqlite"
  host: "localhost"
  port: 5432
  database: "fleet_monitoring"
  username: "${DB_USERNAME}"
  password: "${DB_PASSWORD}"

alerts:
  thresholds:
    temperature_warning: 80
    temperature_critical: 90
    hashrate_drop_percent: 20
    offline_duration_minutes: 5

  email:
    smtp_server: "smtp.gmail.com"
    smtp_port: 587
    username: "${EMAIL_USERNAME}"
    password: "${EMAIL_PASSWORD}"
    recipients:
      - "admin@mining-operation.com"

  slack:
    webhook_url: "${SLACK_WEBHOOK_URL}"
    channel: "#mining-alerts"

logging:
  level: "INFO"
  file: "fleet_monitoring.log"
  max_size_mb: 100
  backup_count: 5
```

5.3 Security Considerations

API Security

```python
import hashlib
import hmac
import time
from typing import Optional

class APISecurityManager:
    def __init__(self, secret_key: str):
        self.secret_key = secret_key.encode()

    def generate_signature(self, data: str, timestamp: str) -> str:
        """Generate HMAC signature for API requests"""
        message = f"{timestamp}:{data}".encode()
        signature = hmac.new(self.secret_key, message, hashlib.sha256)
        return signature.hexdigest()

    def verify_signature(self, data: str, timestamp: str, signature: str) -> bool:
        """Verify HMAC signature"""
        expected_signature = self.generate_signature(data, timestamp)
        return hmac.compare_digest(expected_signature, signature)

    def is_timestamp_valid(self, timestamp: str, max_age_seconds: int = 300) -> bool:
        """Check if timestamp is within acceptable range"""
        try:
            request_time = float(timestamp)
            current_time = time.time()
            return abs(current_time - request_time) <= max_age_seconds
        except (ValueError, TypeError):
            return False

# Network security for miner access
class MinerNetworkSecurity:
    def __init__(self, allowed_networks: List[str]):
        self.allowed_networks = allowed_networks

    def is_ip_allowed(self, ip_address: str) -> bool:
        """Check if IP is in allowed network ranges"""
        import ipaddress

        try:
            ip = ipaddress.ip_address(ip_address)
            for network in self.allowed_networks:
                if ip in ipaddress.ip_network(network):
                    return True
            return False
        except ValueError:
            return False
```

5.4 Performance Optimization

Connection Pooling

```python
import aiohttp
import asyncio
from aiohttp import TCPConnector

class OptimizedAPIClient:
    def __init__(self, max_connections: int = 100, max_connections_per_host: int = 10):
        self.connector = TCPConnector(
            limit=max_connections,
            limit_per_host=max_connections_per_host,
            ttl_dns_cache=300,
            use_dns_cache=True,
        )
        self.session = None

    async def __aenter__(self):
        self.session = aiohttp.ClientSession(
            connector=self.connector,
            timeout=aiohttp.ClientTimeout(total=30)
        )
        return self.session

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        if self.session:
            await self.session.close()

# Usage
async def optimized_fleet_monitoring():
    async with OptimizedAPIClient() as session:
        tasks = []
        for miner_ip in miner_ips:
            task = poll_miner_optimized(session, miner_ip)
            tasks.append(task)

        results = await asyncio.gather(*tasks, return_exceptions=True)
        return results
```

6. Troubleshooting Guide

6.1 Common Issues

Vnish API Connection Issues

```python
def diagnose_vnish_connection(miner_ip: str) -> Dict:
    """Diagnose Vnish API connection issues"""
    diagnosis = {
        "miner_ip": miner_ip,
        "tests": {}
    }

    # Test 1: Ping connectivity
    import subprocess
    try:
        result = subprocess.run(
            ["ping", "-c", "1", "-W", "3", miner_ip],
            capture_output=True,
            text=True,
            timeout=5
        )
        diagnosis["tests"]["ping"] = result.returncode == 0
    except subprocess.TimeoutExpired:
        diagnosis["tests"]["ping"] = False

    # Test 2: HTTP connectivity
    try:
        response = requests.get(f"http://{miner_ip}/", timeout=5)
        diagnosis["tests"]["http"] = response.status_code in [200, 404]
    except requests.RequestException:
        diagnosis["tests"]["http"] = False

    # Test 3: API endpoint availability
    try:
        response = requests.get(f"http://{miner_ip}/docs/", timeout=5)
        diagnosis["tests"]["api_docs"] = response.status_code == 200
    except requests.RequestException:
        diagnosis["tests"]["api_docs"] = False

    # Test 4: JSON response validity
    try:
        response = requests.get(f"http://{miner_ip}/summary", timeout=5)
        data = response.json()
        diagnosis["tests"]["json_response"] = isinstance(data, dict)
    except (requests.RequestException, json.JSONDecodeError):
        diagnosis["tests"]["json_response"] = False

    return diagnosis
```

Prohashing API Issues

```python
def diagnose_prohashing_connection(api_key: str) -> Dict:
    """Diagnose Prohashing API connection issues"""
    diagnosis = {
        "api_key_provided": bool(api_key),
        "tests": {}
    }

    if not api_key:
        return diagnosis

    api = ProhashingAPI(api_key)

    # Test 1: Service health
    diagnosis["tests"]["service_health"] = api.health_check()

    # Test 2: Authentication
    try:
        status = api.get_status()
        diagnosis["tests"]["authentication"] = status is not None
    except Exception as e:
        diagnosis["tests"]["authentication"] = False
        diagnosis["auth_error"] = str(e)

    # Test 3: Data freshness
    diagnosis["tests"]["data_fresh"] = api.is_data_fresh()

    # Test 4: Profitability data
    try:
        profitability = api.get_profitability()
        diagnosis["tests"]["profitability_data"] = bool(profitability)
        diagnosis["algorithms_available"] = len(profitability) if profitability else 0
    except Exception as e:
        diagnosis["tests"]["profitability_data"] = False
        diagnosis["profitability_error"] = str(e)

    return diagnosis
```

6.2 Performance Monitoring

System Health Checks

```
import psutil
import asyncio

class SystemHealthMonitor:
    def __init__(self):
        self.metrics = {}

    def get_system_metrics(self) -> Dict:
        """Get current system performance metrics"""
        return {
            "cpu_percent": psutil.cpu_percent(interval=1),
            "memory_percent": psutil.virtual_memory().percent,
            "disk_percent": psutil.disk_usage('/').percent,
            "network_connections": len(psutil.net_connections()),
            "process_count": len(psutil.pids()),
            "load_average": psutil.getloadavg() if hasattr(psutil, 'getloadavg') else N
one
        }

    async def monitor_api_performance(self, api_calls: List[Callable]) -> Dict:
        """Monitor API call performance"""
        performance_metrics = {}

        for api_call in api_calls:
            start_time = time.time()
            try:
                await api_call()
                success = True
                error = None
            except Exception as e:
                success = False
                error = str(e)

            end_time = time.time()

            performance_metrics[api_call.__name__] = {
                "duration_seconds": end_time - start_time,
                "success": success,
                "error": error
            }

        return performance_metrics
```

7. Conclusion

This comprehensive guide provides the technical foundation for integrating Vnish Hashcore Toolkit and Prohashing.com APIs into a robust cryptocurrency mining fleet monitoring system. The implementation covers:

Key Integration Points

1. **Vnish API**: Direct miner telemetry access via HTTP REST API

2. **Prohashing API**: Pool performance monitoring via WAMP/REST protocols
3. **Data Synchronization**: Efficient polling and real-time data patterns
4. **Error Handling**: Circuit breakers, retry logic, and graceful degradation
5. **Fleet Management**: Scalable monitoring for 6-60 mining machines

Implementation Priorities

1. Start with basic connectivity and data collection
2. Implement robust error handling and monitoring
3. Add alerting and notification systems
4. Optimize for performance and scalability
5. Enhance with advanced analytics and reporting

Next Steps

1. Set up development environment with sample miners
2. Implement core API integration classes
3. Build data storage and persistence layer
4. Create monitoring dashboard and alerting system
5. Deploy and test with production fleet

This guide serves as a complete reference for building a production-ready mining fleet monitoring system with proper error handling, security considerations, and scalability patterns.