Claude Code Master Query - Trading System

AIAlgoTradeHits.com Configuration & Deployment Guide

Version: 4.0
Last Updated: January 2026
Location: `Trading/masterquery.md`

TABLE OF CONTENTS

CONFIGURATION MANAGEMENT PRINCIPLES

Core Rules

1. **NEVER hardcode any variables** - All values must come from config files or environment variables
2. **Centralized config management** - All `.env` files managed through admin panel
3. **Environment separation** - Strict Dev/Prod isolation with fixed URL aliases
4. **Admin control** - All configuration changes require admin authentication

---

📂 CONFIG FILE STRUCTURE

Directory Layout

```
/config/
├── .env.dev              # Development environment
├── .env.prod             # Production environment
├── .env.shared           # Shared across all environments
├── config.yaml           # Main configuration file
├── secrets.yaml          # Encrypted secrets (gitignored)
└── admin/
    ├── config_editor.py  # Admin config editor
    └── permissions.yaml  # Admin access control
```

Main Config File (`config/config.yaml`)

yaml

```yaml
# ============================================
# AIALGOTRADEHITS MASTER CONFIGURATION
# ============================================
# DO NOT HARDCODE VALUES - Reference .env files

app:
  name: ${APP_NAME}
  version: ${APP_VERSION}
  environment: ${ENVIRONMENT}  # 'development' or 'production'

# Cloud Run Services - Fixed Aliases
cloud_run:
  dev:
    alias: "AIAlgoTradeHitsDev"
    url: ${DEV_CLOUD_RUN_URL}
    service_name: ${DEV_SERVICE_NAME}
    min_instances: ${DEV_MIN_INSTANCES}
    max_instances: ${DEV_MAX_INSTANCES}
  prod:
    alias: "AIAlgoTradeHits"
    url: ${PROD_CLOUD_RUN_URL}
    service_name: ${PROD_SERVICE_NAME}
    min_instances: ${PROD_MIN_INSTANCES}
    max_instances: ${PROD_MAX_INSTANCES}

# API Configuration
apis:
  twelvedata:
    base_url: ${TWELVEDATA_BASE_URL}
    api_key: ${TWELVEDATA_API_KEY}
    rate_limit: ${TWELVEDATA_RATE_LIMIT}
    batch_size: ${TWELVEDATA_BATCH_SIZE}

  kraken:
    ws_url: ${KRAKEN_WS_URL}
    rest_url: ${KRAKEN_REST_URL}
    api_key: ${KRAKEN_API_KEY}
    api_secret: ${KRAKEN_API_SECRET}
    nonce_window: ${KRAKEN_NONCE_WINDOW}

  coinmarketcap:
    base_url: ${CMC_BASE_URL}
    api_key: ${CMC_API_KEY}
```

```
    rate_limit: ${CMC_RATE_LIMIT}

  finnhub:
    base_url: ${FINNHUB_BASE_URL}
    api_key: ${FINNHUB_API_KEY}

# GCP Configuration
gcp:
  project_id: ${GCP_PROJECT_ID}
  region: ${GCP_REGION}
  bigquery:
    dataset: ${BIGQUERY_DATASET}
    location: ${BIGQUERY_LOCATION}

# AI Configuration
ai:
  vertex:
    model: ${VERTEX_AI_MODEL}
    fallback_model: ${VERTEX_AI_FALLBACK}
    temperature: ${AI_TEMPERATURE}
    max_tokens: ${AI_MAX_TOKENS}

  anthropic:
    api_key: ${ANTHROPIC_API_KEY}
    model: ${CLAUDE_MODEL}

# Performance Settings
performance:
  parallel_workers: ${PARALLEL_WORKERS}
  batch_size: ${BATCH_SIZE}
  enable_caching: ${ENABLE_CACHING}
  cache_ttl: ${CACHE_TTL}
```

🌍 ENVIRONMENT FILES

Development Environment (`.env.dev`)

bash

```
# ============================================
# DEVELOPMENT ENVIRONMENT CONFIGURATION
# ============================================
# Admin-editable via /admin/config panel

# App Settings
ENVIRONMENT=development
APP_NAME=AIAlgoTradeHits-Dev
APP_VERSION=4.0.0
DEBUG=true
LOG_LEVEL=DEBUG

# Cloud Run - Development
DEV_CLOUD_RUN_URL=https://aialgotradehitsdev-xxxxx-uc.a.run.app
DEV_SERVICE_NAME=AIAlgoTradeHitsDev
DEV_MIN_INSTANCES=0
DEV_MAX_INSTANCES=5
DEV_MEMORY=512Mi
DEV_CPU=1
DEV_TIMEOUT=300

# GCP - Development
GCP_PROJECT_ID=cryptobot-462709
GCP_REGION=us-central1
BIGQUERY_DATASET=crypto_trading_data_dev
BIGQUERY_LOCATION=US

# API Keys (Dev Tier)
TWELVEDATA_API_KEY=${TWELVEDATA_API_KEY_DEV}
TWELVEDATA_BASE_URL=https://api.twelvedata.com
TWELVEDATA_RATE_LIMIT=8
TWELVEDATA_BATCH_SIZE=8

KRAKEN_WS_URL=wss://ws.kraken.com/
KRAKEN_REST_URL=https://api.kraken.com
KRAKEN_NONCE_WINDOW=5000

CMC_BASE_URL=https://pro-api.coinmarketcap.com
CMC_API_KEY=${CMC_API_KEY_DEV}
CMC_RATE_LIMIT=30

FINNHUB_BASE_URL=https://finnhub.io/api/v1
FINNHUB_API_KEY=${FINNHUB_API_KEY_DEV}
```

```
# AI Configuration
ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY_DEV}
CLAUDE_MODEL=claude-sonnet-4-20250514
VERTEX_AI_MODEL=gemini-2.5-pro
VERTEX_AI_FALLBACK=gemini-2.0-pro
AI_TEMPERATURE=0.1
AI_MAX_TOKENS=8192

# Performance (Lower for Dev)
PARALLEL_WORKERS=3
BATCH_SIZE=4
ENABLE_CACHING=true
CACHE_TTL=300
```

Production Environment (`.env.prod`)

bash

```
# ============================================
# PRODUCTION ENVIRONMENT CONFIGURATION
# ============================================
# Admin-editable via /admin/config panel
# ⚠️ Changes require approval workflow

# App Settings
ENVIRONMENT=production
APP_NAME=AIAlgoTradeHits
APP_VERSION=4.0.0
DEBUG=false
LOG_LEVEL=INFO

# Cloud Run - Production
PROD_CLOUD_RUN_URL=https://aialgotradehits-xxxxx-uc.a.run.app
PROD_SERVICE_NAME=AIAlgoTradeHits
PROD_MIN_INSTANCES=1
PROD_MAX_INSTANCES=100
PROD_MEMORY=2Gi
PROD_CPU=2
PROD_TIMEOUT=540

# GCP - Production
GCP_PROJECT_ID=cryptobot-462709
GCP_REGION=us-central1
BIGQUERY_DATASET=crypto_trading_data
BIGQUERY_LOCATION=US

# API Keys (Production Tier)
TWELVEDATA_API_KEY=${TWELVEDATA_API_KEY_PROD}
TWELVEDATA_BASE_URL=https://api.twelvedata.com
TWELVEDATA_RATE_LIMIT=800
TWELVEDATA_BATCH_SIZE=8

KRAKEN_WS_URL=wss://ws.kraken.com/
KRAKEN_REST_URL=https://api.kraken.com
KRAKEN_NONCE_WINDOW=10000

CMC_BASE_URL=https://pro-api.coinmarketcap.com
CMC_API_KEY=${CMC_API_KEY_PROD}
CMC_RATE_LIMIT=120

FINNHUB_BASE_URL=https://finnhub.io/api/v1
```

```
FINNHUB_API_KEY=${FINNHUB_API_KEY_PROD}

# AI Configuration
ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY_PROD}
CLAUDE_MODEL=claude-sonnet-4-20250514
VERTEX_AI_MODEL=gemini-2.5-pro
VERTEX_AI_FALLBACK=gemini-2.0-pro
AI_TEMPERATURE=0.1
AI_MAX_TOKENS=8192

# Performance (Optimized for Prod)
PARALLEL_WORKERS=10
BATCH_SIZE=8
ENABLE_CACHING=true
CACHE_TTL=3600
```

Shared Configuration (`.env.shared`)

bash

```
# ================================================
# SHARED CONFIGURATION (All Environments)
# ================================================

# BigQuery Tables (Consistent naming)
TABLE_CRYPTO_DAILY=crypto_daily
TABLE_STOCKS_DAILY=stocks_daily
TABLE_ETF_DAILY=etfs_daily
TABLE_FOREX_DAILY=forex_daily
TABLE_INDICES_DAILY=indices_daily
TABLE_COMMODITIES_DAILY=commodities_daily

# Asset Types
SUPPORTED_ASSETS=crypto,stocks,etfs,forex,indices,commodities

# Timeframes
SUPPORTED_TIMEFRAMES=1min,5min,15min,30min,hourly,daily,weekly

# Technical Indicators
DEFAULT_RSI_PERIOD=14
DEFAULT_MACD_FAST=12
DEFAULT_MACD_SLOW=26
DEFAULT_MACD_SIGNAL=9
DEFAULT_BB_PERIOD=20
DEFAULT_BB_STD=2

# ================================================
# FAULT TOLERANCE CONFIGURATION
# ================================================

# Retry Settings
RETRY_MAX_ATTEMPTS=5
RETRY_BASE_DELAY=1.0
RETRY_MAX_DELAY=60.0
RETRY_EXPONENTIAL_BASE=2.0
RETRY_JITTER_FACTOR=0.1

# Circuit Breaker Settings
CIRCUIT_FAILURE_THRESHOLD=5
CIRCUIT_SUCCESS_THRESHOLD=3
CIRCUIT_TIMEOUT=60.0
CIRCUIT_HALF_OPEN_MAX=3
```

```
# Data Validation
MAX_CANDLE_MOVE_PCT=50.0

# Alerting Thresholds
ALERT_CONSECUTIVE_FAILURES=3
ALERT_ERROR_RATE_PCT=10.0
ALERT_GAP_HOURS=4.0

# Provider-Specific Thresholds
TWELVEDATA_DAILY_CREDITS=1152000
TWELVEDATA_TARGET_UTILIZATION=0.50
KRAKEN_COUNTER_MAX=20
KRAKEN_COUNTER_DECAY_PRO=1.0
CMC_REQUESTS_PER_MINUTE=120
FINNHUB_REQUESTS_PER_MINUTE=60
```

🚀 CLOUD RUN DEPLOYMENT

URL Alias System

```
# Fixed Aliases (Never Change)
DEV_ALIAS  = "AIAlgoTradeHitsDev"    → Points to DEV_CLOUD_RUN_URL
PROD_ALIAS = "AIAlgoTradeHits"       → Points to PROD_CLOUD_RUN_URL

# Actual URLs (Editable in .env)
DEV_CLOUD_RUN_URL  = https://aialgotradehitsdev-xxxxx-uc.a.run.app
PROD_CLOUD_RUN_URL = https://aialgotradehits-yyyyy-uc.a.run.app
```

Deploy Script (`scripts/deploy.py`)

```
python
```

```python
#!/usr/bin/env python3
"""
Cloud Run Deployment Script
Reads all configuration from .env files - NO HARDCODING
"""
import os
from pathlib import Path
from dotenv import load_dotenv

class CloudRunDeployer:
    def __init__(self, environment: str):
        self.environment = environment
        self._load_config()

    def _load_config(self):
        """Load configuration from appropriate .env file"""
        config_path = Path(__file__).parent.parent / 'config'

        # Load shared config first
        load_dotenv(config_path / '.env.shared')

        # Load environment-specific config
        env_file = f'.env.{self.environment}'
        load_dotenv(config_path / env_file, override=True)

        # Set instance variables from environment
        self.project_id = os.getenv('GCP_PROJECT_ID')
        self.region = os.getenv('GCP_REGION')
        self.service_name = os.getenv(f'{self.environment.upper()}_SERVICE_NAME')
        self.min_instances = os.getenv(f'{self.environment.upper()}_MIN_INSTANCES')
        self.max_instances = os.getenv(f'{self.environment.upper()}_MAX_INSTANCES')
        self.memory = os.getenv(f'{self.environment.upper()}_MEMORY')
        self.cpu = os.getenv(f'{self.environment.upper()}_CPU')
        self.timeout = os.getenv(f'{self.environment.upper()}_TIMEOUT')

    def deploy(self):
        """Deploy to Cloud Run using config values"""
        cmd = f"""
        gcloud run deploy {self.service_name} \\
            --project={self.project_id} \\
            --region={self.region} \\
            --memory={self.memory} \\
            --cpu={self.cpu} \\
```

```
        --timeout={self.timeout} \\
        --min-instances={self.min_instances} \\
        --max-instances={self.max_instances} \\
        --set-env-vars="ENVIRONMENT={self.environment}" \\
        --source=.
    """
    os.system(cmd)
```

🔧 ADMIN CONFIG EDITOR

Config Editor (`config/admin/config_editor.py`)

```
python
```

```
#!/usr/bin/env python3
"""
Admin Configuration Editor
All changes are logged and require authentication
"""
import os
from pathlib import Path
from datetime import datetime
from typing import Dict, Optional

class ConfigEditor:
    def __init__(self, admin_user: str):
        self.admin_user = admin_user
        self.config_path = Path(__file__).parent.parent
        self.audit_log = self.config_path / 'admin' / 'audit.log'

    def get_config(self, environment: str) -> Dict[str, str]:
        """Read current configuration"""
        env_file = self.config_path / f'.env.{environment}'
        config = {}

        with open(env_file) as f:
            for line in f:
                line = line.strip()
                if line and not line.startswith('#') and '=' in line:
                    key, value = line.split('=', 1)
                    config[key] = value

        return config

    def update_config(self, environment: str, key: str, value: str,
                      require_confirmation: bool = True) -> bool:
        """Update a configuration value with audit logging"""
        current_config = self.get_config(environment)
        old_value = current_config.get(key, 'NOT_SET')

        if require_confirmation:
            confirm = input(f"Change {key} from '{old_value}' to '{value}'? (yes/no): ")
            if confirm.lower() != 'yes':
                print("Change cancelled")
                return False

        current_config[key] = value
```

```
        self._write_config(environment, current_config)
        self._log_change(environment, key, old_value, value)

        print(f"✅ Updated {key} in {environment}")
        return True

    def _write_config(self, environment: str, config: Dict[str, str]):
        """Write config back to .env file"""
        env_file = self.config_path / f'.env.{environment}'

        with open(env_file, 'w') as f:
            f.write(f"# {environment.upper()} ENVIRONMENT CONFIGURATION\n")
            f.write(f"# Last modified: {datetime.utcnow().isoformat()}\n")
            f.write(f"# Modified by: {self.admin_user}\n\n")

            for key, value in sorted(config.items()):
                f.write(f"{key}={value}\n")

    def _log_change(self, environment: str, key: str, old_value: str, new_value: str):
        """Log configuration change for audit"""
        with open(self.audit_log, 'a') as f:
            f.write(f"{datetime.utcnow().isoformat()}|{self.admin_user}|{environment}|{key}|{old_value}|{new_value}\n")
```

🔄 EMA CYCLE DETECTION

Rise/Fall Cycle Logic

```
sql

-- Rise Cycle Detection
in_rise_cycle = (ema_12 > ema_26)

-- Rise Cycle Start (crossover)
rise_cycle_start = (ema_12 > ema_26) AND (LAG(ema_12) <= LAG(ema_26))

-- Fall Cycle Start (crossunder)
fall_cycle_start = (ema_12 < ema_26) AND (LAG(ema_12) >= LAG(ema_26))
```

Supporting Confirmations

```
sql
```

```
-- Volume Confirmation
volume_confirmed = (volume > AVG(volume) OVER (ORDER BY datetime ROWS 20 PRECEDING) * 1.2)

-- RSI Confirmation
rsi_bullish = (rsi_14 > 50)
rsi_bearish = (rsi_14 < 50)

-- Full Rise Signal
rise_signal = rise_cycle_start AND volume_confirmed AND rsi_bullish
```

📊 TREND REGIME CLASSIFICATION

Regime States

```
sql

trend_regime = CASE
  WHEN close > sma_50 AND sma_50 > sma_200 AND adx > 25 THEN 'STRONG_UPTREND'
  WHEN close > sma_50 AND close > sma_200 THEN 'WEAK_UPTREND'
  WHEN close < sma_50 AND sma_50 < sma_200 AND adx > 25 THEN 'STRONG_DOWNTREND'
  WHEN close < sma_50 AND close < sma_200 THEN 'WEAK_DOWNTREND'
  ELSE 'CONSOLIDATION'
END
```

Golden/Death Cross Detection

```
sql

-- Golden Cross (Bullish)
golden_cross = (sma_50 > sma_200) AND (LAG(sma_50) <= LAG(sma_200))

-- Death Cross (Bearish)
death_cross = (sma_50 < sma_200) AND (LAG(sma_50) >= LAG(sma_200))
```

🧠 ML INDICATOR FRAMEWORK BY TIMEFRAME

Timeframe-Specific Indicator Counts

Timeframe	Indicators	Purpose	Latency Target
Daily	24 (full)	Strategic screening	<5 seconds
Hourly	12	Cycle timing	<1 second
5-Minute	8	Trade execution	<200ms
1-Minute	5	Scalping (optional)	<50ms

Daily (24 Indicators)

Momentum: RSI, MACD, ROC, Stoch_K, Stoch_D, MFI
Trend: SMA_20/50/200, EMA_12/20/26/50/200, Ichimoku_Tenkan/Kijun
Volatility: ATR, BB_Upper/Middle/Lower
Strength: ADX, Plus_DI, Minus_DI
Flow: MFI, CMF

Hourly (12 Indicators)

```yaml
cycle_detection: [EMA_9, EMA_21]
momentum: [RSI_14, MACD, MACD_Histogram]
volume: [Volume_Ratio, VWAP]
volatility: [ATR_14, BB_Percent_B]
trend_context: [SMA_50, ADX]
flow: [MFI]
```

5-Minute (8 Indicators)

```yaml
signal: [EMA_9, EMA_21]
momentum: [RSI_14, MACD_Histogram]
volume: [Volume_Ratio, VWAP]
risk: [ATR_14, Price_vs_VWAP]
```

1-Minute (5 Indicators)

```yaml
signal: [EMA_5, EMA_13]
momentum: [RSI_7]
volume: [Volume_Spike]
reference: [VWAP_Distance_Pct]
```

FAULT TOLERANT ROUTINE

Overview

This section defines the fault tolerance patterns and error recovery mechanisms for all API integrations and data processing operations in the AIAlgoTradeHits platform.

Retry with Exponential Backoff

Core Implementation

```python
```

```python
import time
import random
import asyncio
import os
from functools import wraps
from typing import Callable, Type, Tuple
import logging

logger = logging.getLogger(__name__)

class RetryConfig:
    """Centralized retry configuration - values from .env files"""
    MAX_RETRIES = int(os.getenv('RETRY_MAX_ATTEMPTS', 5))
    BASE_DELAY = float(os.getenv('RETRY_BASE_DELAY', 1.0))
    MAX_DELAY = float(os.getenv('RETRY_MAX_DELAY', 60.0))
    EXPONENTIAL_BASE = float(os.getenv('RETRY_EXPONENTIAL_BASE', 2.0))
    JITTER_FACTOR = float(os.getenv('RETRY_JITTER_FACTOR', 0.1))

class RetryableHTTPError(Exception):
    """Exception for retryable HTTP errors"""
    def __init__(self, message: str, status_code: int = None):
        super().__init__(message)
        self.status_code = status_code

class CircuitOpenError(Exception):
    """Exception when circuit breaker is open"""
    pass

def exponential_backoff_retry(
    max_retries: int = None,
    base_delay: float = None,
    max_delay: float = None,
    retryable_exceptions: Tuple[Type[Exception], ...] = (Exception,),
    retryable_status_codes: Tuple[int, ...] = (429, 500, 502, 503, 504)
):
    """
    Decorator for fault-tolerant API calls with exponential backoff.

    Features:
    - Exponential delay increase: delay = base * (2 ^ attempt)
    - Jitter to prevent thundering herd
    - Configurable max delay cap
    - Selective retry based on exception type or HTTP status
```

```
    """
    max_retries = max_retries or RetryConfig.MAX_RETRIES
    base_delay = base_delay or RetryConfig.BASE_DELAY
    max_delay = max_delay or RetryConfig.MAX_DELAY

    def decorator(func: Callable):
        @wraps(func)
        async def async_wrapper(*args, **kwargs):
            last_exception = None

            for attempt in range(max_retries + 1):
                try:
                    result = await func(*args, **kwargs)

                    # Check for retryable HTTP status codes
                    if hasattr(result, 'status_code'):
                        if result.status_code in retryable_status_codes:
                            raise RetryableHTTPError(
                                f"HTTP {result.status_code}",
                                status_code=result.status_code
                            )
                    return result

                except retryable_exceptions as e:
                    last_exception = e

                    if attempt == max_retries:
                        logger.error(f"Max retries ({max_retries}) exceeded for {func.__name__}: {e}")
                        raise

                    # Calculate delay with exponential backoff + jitter
                    delay = min(
                        base_delay * (RetryConfig.EXPONENTIAL_BASE ** attempt),
                        max_delay
                    )
                    jitter = delay * RetryConfig.JITTER_FACTOR * random.random()
                    total_delay = delay + jitter

                    logger.warning(
                        f"Attempt {attempt + 1}/{max_retries + 1} failed for {func.__name__}. "
                        f"Retrying in {total_delay:.2f}s. Error: {e}"
                    )

                    await asyncio.sleep(total_delay)
```

```
    raise last_exception

@wraps(func)
def sync_wrapper(*args, **kwargs):
    last_exception = None

    for attempt in range(max_retries + 1):
        try:
            result = func(*args, **kwargs)

            if hasattr(result, 'status_code'):
                if result.status_code in retryable_status_codes:
                    raise RetryableHTTPError(
                        f"HTTP {result.status_code}",
                        status_code=result.status_code
                    )
            return result

        except retryable_exceptions as e:
            last_exception = e

            if attempt == max_retries:
                logger.error(f"Max retries exceeded for {func.__name__}: {e}")
                raise

            delay = min(
                base_delay * (RetryConfig.EXPONENTIAL_BASE ** attempt),
                max_delay
            )
            jitter = delay * RetryConfig.JITTER_FACTOR * random.random()
            total_delay = delay + jitter

            logger.warning(
                f"Attempt {attempt + 1}/{max_retries + 1} failed. "
                f"Retrying in {total_delay:.2f}s. Error: {e}"
            )

            time.sleep(total_delay)

    raise last_exception

# Return appropriate wrapper based on function type
if asyncio.iscoroutinefunction(func):
```

```
            return async_wrapper
        return sync_wrapper

    return decorator
```

⚠️ Error Classification & Recovery Matrix

Error Type	HTTP Code	Detection Method	Recovery Action	Max Retries
Rate Limit	429	Response status	Exponential backoff + respect Retry-After header	5
Server Error	500	Response status	Retry with backoff	3
Bad Gateway	502	Response status	Retry with backoff	3
Service Unavailable	503	Response status	Retry with longer delay	5
Gateway Timeout	504	Response status	Retry with increased timeout	3
Connection Error	N/A	Exception	Retry with backoff	5
Timeout	N/A	Timeout exception	Retry with 2x timeout	3
Invalid Data	400	Response status	Log & skip (no retry)	0
Auth Error	401/403	Response status	Alert & halt	0
Not Found	404	Response status	Log & skip	0
Duplicate Data	N/A	Primary key conflict	Skip insert, log	0
Schema Mismatch	N/A	Type error	Transform or reject	0
Kraken Invalid Nonce	N/A	EAPI:Invalid nonce	Resync nonce, retry	3
Kraken API Limit	N/A	EAPI:Rate limit	Wait for counter decay	5

Provider-Specific Thresholds (Tuned)

python

```
PROVIDER_THRESHOLDS = {
    'twelvedata': {
        'max_retries': 5,
        'base_delay': 1.0,
        'max_delay': 60.0,
        'rate_limit_buffer': 0.9,  # Use 90% of limit
        'batch_size': 8,
        'daily_credit_target': 576000,  # 50% of 1.152M
    },
    'kraken': {
        'max_retries': 3,
        'base_delay': 0.5,
        'max_delay': 30.0,
        'counter_max': 20,
        'counter_decay': 1.0,  # Pro tier: 1/sec
        'nonce_window_ms': 10000,
    },
    'coinmarketcap': {
        'max_retries': 5,
        'base_delay': 2.0,
        'max_delay': 120.0,
        'requests_per_minute': 120,
    },
    'finnhub': {
        'max_retries': 5,
        'base_delay': 1.0,
        'max_delay': 60.0,
        'requests_per_minute': 60,
    },
    'bigquery': {
        'max_retries': 3,
        'base_delay': 2.0,
        'max_delay': 30.0,
    }
}
```

🔌 Circuit Breaker Pattern

python

```python
import time
from enum import Enum
from dataclasses import dataclass
from threading import Lock

class CircuitState(Enum):
    CLOSED = "closed"      # Normal operation
    OPEN = "open"          # Failing, reject calls
    HALF_OPEN = "half_open"  # Testing recovery

@dataclass
class CircuitBreakerConfig:
    """Configuration from .env files"""
    failure_threshold: int = int(os.getenv('CIRCUIT_FAILURE_THRESHOLD', 5))
    success_threshold: int = int(os.getenv('CIRCUIT_SUCCESS_THRESHOLD', 3))
    timeout: float = float(os.getenv('CIRCUIT_TIMEOUT', 60.0))
    half_open_max_calls: int = int(os.getenv('CIRCUIT_HALF_OPEN_MAX', 3))

class CircuitBreaker:
    """
    Circuit breaker to prevent cascading failures.

    States:
    - CLOSED: Normal operation, tracking failures
    - OPEN: Too many failures, rejecting all calls
    - HALF_OPEN: Testing if service recovered
    """

    def __init__(self, name: str, config: CircuitBreakerConfig = None):
        self.name = name
        self.config = config or CircuitBreakerConfig()
        self.state = CircuitState.CLOSED
        self.failure_count = 0
        self.success_count = 0
        self.last_failure_time = None
        self.half_open_calls = 0
        self._lock = Lock()

    def can_execute(self) -> bool:
        """Check if call should be allowed"""
        with self._lock:
            if self.state == CircuitState.CLOSED:
                return True
```

```
        if self.state == CircuitState.OPEN:
            # Check if timeout has passed
            if time.time() - self.last_failure_time >= self.config.timeout:
                self.state = CircuitState.HALF_OPEN
                self.half_open_calls = 0
                logger.info(f"Circuit {self.name} transitioning to HALF_OPEN")
                return True
            return False

        if self.state == CircuitState.HALF_OPEN:
            if self.half_open_calls < self.config.half_open_max_calls:
                self.half_open_calls += 1
                return True
            return False

        return False

    def record_success(self):
        """Record successful call"""
        with self._lock:
            if self.state == CircuitState.HALF_OPEN:
                self.success_count += 1
                if self.success_count >= self.config.success_threshold:
                    self.state = CircuitState.CLOSED
                    self.failure_count = 0
                    self.success_count = 0
                    logger.info(f"Circuit {self.name} CLOSED - service recovered")
            else:
                self.failure_count = max(0, self.failure_count - 1)

    def record_failure(self):
        """Record failed call"""
        with self._lock:
            self.failure_count += 1
            self.last_failure_time = time.time()

            if self.state == CircuitState.HALF_OPEN:
                self.state = CircuitState.OPEN
                self.success_count = 0
                logger.warning(f"Circuit {self.name} OPEN - recovery failed")
            elif self.failure_count >= self.config.failure_threshold:
                self.state = CircuitState.OPEN
                logger.warning(f"Circuit {self.name} OPEN - threshold exceeded")
```

```python
# Circuit breakers per API provider
CIRCUIT_BREAKERS = {
    'twelvedata': CircuitBreaker('twelvedata'),
    'kraken': CircuitBreaker('kraken'),
    'coinmarketcap': CircuitBreaker('coinmarketcap'),
    'finnhub': CircuitBreaker('finnhub'),
    'bigquery': CircuitBreaker('bigquery'),
}
```

Kraken Nonce-Based Rate Limiting Handler

python

```
import time
import hmac
import hashlib
import base64
import urllib.parse
from threading import Lock
from typing import Optional, Dict, Any

class KrakenNonceManager:
    """
    Manages Kraken API nonce generation and rate limiting.

    Kraken specifics:
    - Nonce must be strictly increasing (millisecond timestamp)
    - Cannot reset nonce to lower value
    - Invalid nonce errors can cause temporary bans
    - Counter-based rate limiting with decay
    """

    def __init__(self):
        self.last_nonce = 0
        self.nonce_lock = Lock()
        self.api_counter = 0
        self.last_counter_update = time.time()
        self.counter_lock = Lock()

        # Config from environment
        self.counter_max = int(os.getenv('KRAKEN_COUNTER_MAX', 20))
        self.counter_decay = float(os.getenv('KRAKEN_COUNTER_DECAY_PRO', 1.0))
        self.nonce_window = int(os.getenv('KRAKEN_NONCE_WINDOW', 10000))

    def get_nonce(self) -> str:
        """
        Generate strictly increasing nonce.
        Uses millisecond timestamp with collision prevention.
        """
        with self.nonce_lock:
            current_nonce = int(time.time() * 1000)

            # Ensure strictly increasing
            if current_nonce <= self.last_nonce:
                current_nonce = self.last_nonce + 1
```

```
        self.last_nonce = current_nonce
        return str(current_nonce)

    def _decay_counter(self):
        """Apply counter decay based on time elapsed"""
        current_time = time.time()
        elapsed = current_time - self.last_counter_update
        decay_amount = elapsed * self.counter_decay
        self.api_counter = max(0, self.api_counter - decay_amount)
        self.last_counter_update = current_time

    def can_make_request(self, cost: int = 1) -> bool:
        """Check if request is within rate limits"""
        with self.counter_lock:
            self._decay_counter()
            return (self.api_counter + cost) <= self.counter_max

    def record_request(self, cost: int = 1):
        """Record API request cost"""
        with self.counter_lock:
            self._decay_counter()
            self.api_counter += cost

    def wait_for_capacity(self, cost: int = 1) -> float:
        """Calculate wait time until capacity available"""
        with self.counter_lock:
            self._decay_counter()

            if (self.api_counter + cost) <= self.counter_max:
                return 0.0

            # Calculate time needed for counter to decay
            excess = (self.api_counter + cost) - self.counter_max
            wait_time = excess / self.counter_decay
            return wait_time

    def generate_signature(self, uri_path: str, data: Dict[str, Any],
                           api_secret: str) -> str:
        """
        Generate Kraken API signature.

        Signature = HMAC-SHA512(Path + SHA256(nonce + POST_data), Base64Dec(API_secret))
        """
        postdata = urllib.parse.urlencode(data)
```

```
        encoded = (str(data['nonce']) + postdata).encode()
        message = uri_path.encode() + hashlib.sha256(encoded).digest()

        mac = hmac.new(
            base64.b64decode(api_secret),
            message,
            hashlib.sha512
        )

        return base64.b64encode(mac.digest()).decode()

    def handle_nonce_error(self) -> int:
        """
        Handle EAPI:Invalid nonce error.
        Returns suggested wait time in milliseconds.
        """
        with self.nonce_lock:
            # Jump nonce forward to resync
            self.last_nonce = int(time.time() * 1000) + self.nonce_window
            logger.warning(f"Nonce resync: jumped to {self.last_nonce}")
            return 1000  # Wait 1 second before retry


class KrakenClient:
    """
    Kraken API client with full fault tolerance.
    """

    def __init__(self):
        self.nonce_manager = KrakenNonceManager()
        self.circuit = CIRCUIT_BREAKERS['kraken']
        self.rest_url = os.getenv('KRAKEN_REST_URL', 'https://api.kraken.com')
        self.api_key = os.getenv('KRAKEN_API_KEY')
        self.api_secret = os.getenv('KRAKEN_API_SECRET')

    @exponential_backoff_retry(
        max_retries=3,
        base_delay=0.5,
        retryable_exceptions=(ConnectionError, TimeoutError, RetryableHTTPError)
    )
    async def private_request(self, endpoint: str, data: Dict[str, Any] = None) -> Dict:
        """Make authenticated request to Kraken private API"""

        if not self.circuit.can_execute():
```

```
    raise CircuitOpenError("Kraken circuit breaker is open")

# Check rate limit
wait_time = self.nonce_manager.wait_for_capacity(cost=1)
if wait_time > 0:
    logger.info(f"Kraken rate limit: waiting {wait_time:.2f}s")
    await asyncio.sleep(wait_time)

data = data or {}
data['nonce'] = self.nonce_manager.get_nonce()

uri_path = f'/0/private/{endpoint}'
signature = self.nonce_manager.generate_signature(
    uri_path, data, self.api_secret
)

headers = {
    'API-Key': self.api_key,
    'API-Sign': signature
}

try:
    async with aiohttp.ClientSession() as session:
        async with session.post(
            f"{self.rest_url}{uri_path}",
            headers=headers,
            data=data,
            timeout=aiohttp.ClientTimeout(total=30)
        ) as response:
            result = await response.json()

            # Check for Kraken-specific errors
            if result.get('error'):
                error = result['error'][0] if result['error'] else 'Unknown'

                if 'EAPI:Invalid nonce' in error:
                    wait_ms = self.nonce_manager.handle_nonce_error()
                    await asyncio.sleep(wait_ms / 1000)
                    raise RetryableHTTPError("Invalid nonce", status_code=None)

                if 'EAPI:Rate limit exceeded' in error:
                    raise RetryableHTTPError("Rate limit exceeded", status_code=429)

                if 'EService:Throttled' in error:
```

```
                    # Extract timestamp if available
                    raise RetryableHTTPError("Service throttled", status_code=503)

                raise Exception(f"Kraken API error: {error}")

            self.nonce_manager.record_request(cost=1)
            self.circuit.record_success()
            return result['result']

        except Exception as e:
            self.circuit.record_failure()
            raise

    async def get_ticker(self, pair: str) -> Dict:
        """Get ticker information (public endpoint)"""
        async with aiohttp.ClientSession() as session:
            async with session.get(
                f"{self.rest_url}/0/public/Ticker",
                params={'pair': pair}
            ) as response:
                result = await response.json()
                if result.get('error'):
                    raise Exception(f"Kraken error: {result['error']}")
                return result['result']


# Global Kraken client instance
kraken_client = KrakenClient()
```

📊 Rate Limit Manager (Multi-Provider)

python

```python
from collections import defaultdict
import asyncio

class RateLimitManager:
    """
    Manages API rate limits across all providers.
    Tracks usage and enforces limits proactively.
    """

    def __init__(self):
        self.limits = {
            'twelvedata': {
                'requests_per_minute': int(os.getenv('TWELVEDATA_RATE_LIMIT', 800)),
                'daily_credits': int(os.getenv('TWELVEDATA_DAILY_CREDITS', 1152000)),
                'target_daily': int(os.getenv('TWELVEDATA_DAILY_CREDITS', 1152000)) * 0.5,
            },
            'kraken': {
                'counter_max': int(os.getenv('KRAKEN_COUNTER_MAX', 20)),
                'counter_decay': float(os.getenv('KRAKEN_COUNTER_DECAY_PRO', 1.0)),
            },
            'coinmarketcap': {
                'requests_per_minute': int(os.getenv('CMC_RATE_LIMIT', 120)),
            },
            'finnhub': {
                'requests_per_minute': int(os.getenv('FINNHUB_REQUESTS_PER_MINUTE', 60)),
            }
        }
        self.usage = defaultdict(lambda: {
            'minute': 0,
            'daily': 0,
            'last_reset': time.time(),
            'daily_reset': time.time()
        })
        self._lock = asyncio.Lock()

    async def acquire(self, provider: str, cost: int = 1) -> bool:
        """
        Acquire rate limit tokens before making API call.
        Returns True if allowed, False if should wait.
        """
        async with self._lock:
            self._reset_if_needed(provider)
```

```
        limit_config = self.limits.get(provider, {})
        current_usage = self.usage[provider]

        # Check minute limit
        minute_limit = limit_config.get('requests_per_minute', float('inf'))
        if current_usage['minute'] + cost > minute_limit:
            return False

        # Check daily limit (for TwelveData)
        if provider == 'twelvedata':
            daily_target = limit_config.get('target_daily', float('inf'))
            if current_usage['daily'] + cost > daily_target:
                logger.warning(f"TwelveData daily target ({daily_target}) reached")
                return False

        # Acquire tokens
        current_usage['minute'] += cost
        current_usage['daily'] += cost
        return True

    async def wait_for_capacity(self, provider: str, cost: int = 1):
        """Wait until rate limit capacity is available"""
        while not await self.acquire(provider, cost):
            await asyncio.sleep(1.0)

    def _reset_if_needed(self, provider: str):
        """Reset counters if time periods have passed"""
        current_time = time.time()
        usage = self.usage[provider]

        # Reset minute counter
        if current_time - usage['last_reset'] >= 60:
            usage['minute'] = 0
            usage['last_reset'] = current_time

        # Reset daily counter at midnight UTC
        if current_time - usage['daily_reset'] >= 86400:
            usage['daily'] = 0
            usage['daily_reset'] = current_time

    def handle_429(self, provider: str, retry_after: int = None) -> int:
        """Handle rate limit response from API"""
        logger.warning(f"Rate limit hit for {provider}")
```

```
        # If Retry-After header provided, use it
        if retry_after:
            return retry_after

        # Default backoff times per provider
        default_backoffs = {
            'twelvedata': 60,
            'kraken': 5,
            'coinmarketcap': 60,
            'finnhub': 60,
        }
        return default_backoffs.get(provider, 60)

    def get_utilization(self, provider: str) -> Dict[str, float]:
        """Get current utilization metrics"""
        usage = self.usage[provider]
        limits = self.limits.get(provider, {})

        return {
            'minute_pct': (usage['minute'] / limits.get('requests_per_minute', 1)) * 100,
            'daily_pct': (usage['daily'] / limits.get('daily_credits', 1)) * 100 if 'daily_credits' in limits else None,
        }

# Global rate limit manager
rate_limiter = RateLimitManager()
```

🔍 Gap Detection & Recovery

python

```
from datetime import datetime, timedelta
from google.cloud import bigquery
import json

class GapDetector:
    """
    Detects and manages data gaps in time series data.
    Triggers backfill jobs for missing periods.
    """

    def __init__(self, bq_client: bigquery.Client):
        self.client = bq_client
        self.dataset = os.getenv('BIGQUERY_DATASET')
        self.project = os.getenv('GCP_PROJECT_ID')

    async def detect_gaps(
        self,
        table: str,
        symbol: str,
        timeframe: str,
        lookback_days: int = 7
    ) -> list:
        """
        Detect missing data periods in time series.
        Returns list of gap dictionaries.
        """
        interval_minutes = {
            '1min': 1, '5min': 5, '15min': 15,
            '30min': 30, '1h': 60, '4h': 240, '1day': 1440
        }

        expected_interval = interval_minutes.get(timeframe, 1440)

        query = f"""
        WITH timestamps AS (
            SELECT
                datetime,
                LEAD(datetime) OVER (ORDER BY datetime) as next_datetime
            FROM `{self.project}.{self.dataset}.{table}`
            WHERE symbol = @symbol
              AND datetime >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL @lookback DAY)
            ORDER BY datetime
        )
```

```
    SELECT
        datetime as gap_start,
        next_datetime as gap_end,
        TIMESTAMP_DIFF(next_datetime, datetime, MINUTE) as gap_minutes
    FROM timestamps
    WHERE TIMESTAMP_DIFF(next_datetime, datetime, MINUTE) > @expected_interval * 1.5
    ORDER BY datetime
    """

    job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("symbol", "STRING", symbol),
            bigquery.ScalarQueryParameter("lookback", "INT64", lookback_days),
            bigquery.ScalarQueryParameter("expected_interval", "INT64", expected_interval),
        ]
    )

    results = self.client.query(query, job_config=job_config).result()

    gaps = []
    for row in results:
        gaps.append({
            'start': row.gap_start,
            'end': row.gap_end,
            'duration_minutes': row.gap_minutes
        })

    if gaps:
        logger.warning(f"Detected {len(gaps)} gaps for {symbol} in {table}")
        await self._log_gaps(table, symbol, gaps)

    return gaps

async def _log_gaps(self, table: str, symbol: str, gaps: list):
    """Log detected gaps to data_gaps audit table"""
    rows = [
        {
            'table_name': table,
            'symbol': symbol,
            'gap_start': gap['start'].isoformat(),
            'gap_end': gap['end'].isoformat(),
            'duration_minutes': gap['duration_minutes'],
            'detected_at': datetime.utcnow().isoformat(),
            'status': 'pending_backfill'
```

```
            }
            for gap in gaps
        ]

        errors = self.client.insert_rows_json(
            f"{self.project}.{self.dataset}.data_gaps",
            rows
        )

        if errors:
            logger.error(f"Failed to log gaps: {errors}")

    async def trigger_backfill(self, table: str, symbol: str,
                               start: datetime, end: datetime):
        """Trigger backfill job for specific gap via Cloud Tasks"""
        from google.cloud import tasks_v2

        client = tasks_v2.CloudTasksClient()
        parent = client.queue_path(
            os.getenv('GCP_PROJECT_ID'),
            os.getenv('GCP_REGION'),
            'backfill-queue'
        )

        task = {
            'http_request': {
                'http_method': 'POST',
                'url': f"{os.getenv('PROD_CLOUD_RUN_URL')}/api/admin/backfill",
                'headers': {'Content-Type': 'application/json'},
                'body': json.dumps({
                    'table': table,
                    'symbol': symbol,
                    'start': start.isoformat(),
                    'end': end.isoformat()
                }).encode()
            }
        }

        client.create_task(request={'parent': parent, 'task': task})
        logger.info(f"Backfill task created for {symbol}: {start} to {end}")
```

✅ Data Validation & Integrity

python

```python
from dataclasses import dataclass
from typing import Optional, List, Tuple

@dataclass
class OHLCVRecord:
    """OHLCV data record with validation"""
    symbol: str
    datetime: datetime
    open: float
    high: float
    low: float
    close: float
    volume: float

    def validate(self) -> Tuple[bool, Optional[str]]:
        """
        Validate OHLCV data integrity.
        Returns (is_valid, error_message)
        """
        # Price relationship validation
        if not (self.low <= self.open <= self.high):
            return False, f"Invalid OHLC: open {self.open} outside low-high range"

        if not (self.low <= self.close <= self.high):
            return False, f"Invalid OHLC: close {self.close} outside low-high range"

        if self.high < self.low:
            return False, f"Invalid OHLC: high {self.high} < low {self.low}"

        # Positive values check
        if any(v < 0 for v in [self.open, self.high, self.low, self.close]):
            return False, "Negative price values detected"

        if self.volume < 0:
            return False, "Negative volume detected"

        # Extreme move detection (configurable threshold)
        max_move_pct = float(os.getenv('MAX_CANDLE_MOVE_PCT', 50))
        if self.open > 0:
            move_pct = abs(self.high - self.low) / self.open * 100
            if move_pct > max_move_pct:
                return False, f"Extreme price move: {move_pct:.1f}% exceeds {max_move_pct}%"
```

```python
        return True, None

class DataValidator:
    """Comprehensive data validation for all record types"""

    @staticmethod
    def validate_and_filter(records: List[dict], record_type: str) -> Tuple[List, List]:
        """
        Validate records and separate valid from invalid.
        Returns (valid_records, invalid_records)
        """
        valid = []
        invalid = []

        for record in records:
            try:
                if record_type == 'ohlcv':
                    ohlcv = OHLCVRecord(**record)
                    is_valid, error = ohlcv.validate()
                    if is_valid:
                        valid.append(record)
                    else:
                        invalid.append({'record': record, 'error': error})
                else:
                    # Generic validation for other types
                    valid.append(record)

            except Exception as e:
                invalid.append({'record': record, 'error': str(e)})

        if invalid:
            logger.warning(f"Filtered {len(invalid)} invalid records out of {len(records)}")

        return valid, invalid
```

☁ CLOUD FUNCTIONS INTEGRATION

Fault-Tolerant Cloud Function Template

python

```
"""
Cloud Function template with full fault tolerance.
Deploy as: gcloud functions deploy {function_name} --gen2 --runtime=python311
"""
import functions_framework
import asyncio
import os
import logging
from typing import List, Dict, Any

# Import fault tolerance components
from fault_tolerance import (
    exponential_backoff_retry,
    CIRCUIT_BREAKERS,
    rate_limiter,
    DataValidator,
    GapDetector,
    RetryableHTTPError
)

logger = logging.getLogger(__name__)

# Configuration
PARALLEL_WORKERS = int(os.getenv('PARALLEL_WORKERS', 10))
BATCH_SIZE = int(os.getenv('BATCH_SIZE', 8))
WORKER_ID = int(os.getenv('WORKER_ID', 0))
TOTAL_WORKERS = int(os.getenv('TOTAL_WORKERS', 10))


class FaultTolerantFetcher:
    """Base class for fault-tolerant data fetchers"""

    def __init__(self, provider: str):
        self.provider = provider
        self.circuit = CIRCUIT_BREAKERS.get(provider)
        self.stats = {
            'success': 0,
            'failures': 0,
            'skipped': 0,
            'retries': 0
        }

    async def fetch_batch(self, symbols: List[str]) -> List[Dict]:
```

```
        """Fetch data for a batch of symbols with fault tolerance"""
        results = []

        for symbol in symbols:
            try:
                if self.circuit and not self.circuit.can_execute():
                    logger.warning(f"Circuit open for {self.provider}, skipping {symbol}")
                    self.stats['skipped'] += 1
                    continue

                # Acquire rate limit
                await rate_limiter.wait_for_capacity(self.provider, cost=1)

                # Fetch with retry
                data = await self._fetch_single(symbol)

                # Validate
                valid, invalid = DataValidator.validate_and_filter([data], 'ohlcv')

                if valid:
                    results.extend(valid)
                    self.stats['success'] += 1
                    if self.circuit:
                        self.circuit.record_success()
                else:
                    logger.warning(f"Invalid data for {symbol}: {invalid}")
                    self.stats['failures'] += 1

            except Exception as e:
                logger.error(f"Failed to fetch {symbol}: {e}")
                self.stats['failures'] += 1
                if self.circuit:
                    self.circuit.record_failure()

        return results

    @exponential_backoff_retry(max_retries=3)
    async def _fetch_single(self, symbol: str) -> Dict:
        """Override in subclass for provider-specific fetch logic"""
        raise NotImplementedError


class TwelveDataFetcher(FaultTolerantFetcher):
    """TwelveData-specific fetcher with batch optimization"""
```

```
def __init__(self):
    super().__init__('twelvedata')
    self.api_key = os.getenv('TWELVEDATA_API_KEY')
    self.base_url = os.getenv('TWELVEDATA_BASE_URL')

async def fetch_batch(self, symbols: List[str]) -> List[Dict]:
    """Override to use TwelveData's batch endpoint (8 symbols per request)"""
    results = []

    # Split into batches of 8 (TwelveData limit)
    batches = [symbols[i:i+8] for i in range(0, len(symbols), 8)]

    for batch in batches:
        try:
            if not self.circuit.can_execute():
                logger.warning("TwelveData circuit open")
                break

            # Cost = 1 credit per symbol in batch
            await rate_limiter.wait_for_capacity('twelvedata', cost=len(batch))

            data = await self._fetch_batch_request(batch)

            valid, invalid = DataValidator.validate_and_filter(data, 'ohlcv')
            results.extend(valid)
            self.stats['success'] += len(valid)
            self.stats['failures'] += len(invalid)

            self.circuit.record_success()

        except Exception as e:
            logger.error(f"Batch fetch failed: {e}")
            self.stats['failures'] += len(batch)
            self.circuit.record_failure()

    return results

@exponential_backoff_retry(
    max_retries=5,
    retryable_status_codes=(429, 500, 502, 503, 504)
)
async def _fetch_batch_request(self, symbols: List[str]) -> List[Dict]:
    """Fetch batch from TwelveData API"""
```

```
    import aiohttp

    params = {
        'symbol': ','.join(symbols),
        'interval': '1day',
        'outputsize': 100,
        'apikey': self.api_key
    }

    async with aiohttp.ClientSession() as session:
        async with session.get(
            f"{self.base_url}/time_series",
            params=params,
            timeout=aiohttp.ClientTimeout(total=60)
        ) as response:
            if response.status == 429:
                retry_after = response.headers.get('Retry-After', 60)
                rate_limiter.handle_429('twelvedata', int(retry_after))
                raise RetryableHTTPError("Rate limited", status_code=429)

            response.raise_for_status()
            data = await response.json()

            # Parse TwelveData response format
            return self._parse_response(data, symbols)

def _parse_response(self, data: Dict, symbols: List[str]) -> List[Dict]:
    """Parse TwelveData batch response"""
    results = []

    for symbol in symbols:
        symbol_data = data.get(symbol, {})
        if 'values' in symbol_data:
            for candle in symbol_data['values']:
                results.append({
                    'symbol': symbol,
                    'datetime': candle['datetime'],
                    'open': float(candle['open']),
                    'high': float(candle['high']),
                    'low': float(candle['low']),
                    'close': float(candle['close']),
                    'volume': float(candle.get('volume', 0))
                })
```

```
    return results


@functions_framework.http
def daily_crypto_fetcher(request):
    """
    Cloud Function: Daily crypto data fetcher with fault tolerance.

    Deployed with parallel workers:
    for i in {0..9}; do
      gcloud functions deploy daily-crypto-fetcher-$i \
        --gen2 --runtime=python311 \
        --memory=2GB --timeout=540s \
        --set-env-vars="WORKER_ID=$i,TOTAL_WORKERS=10"
    done
    """
    async def main():
        # Get symbols for this worker
        all_symbols = get_crypto_symbols()  # From config or BigQuery
        worker_symbols = all_symbols[WORKER_ID::TOTAL_WORKERS]

        logger.info(f"Worker {WORKER_ID}/{TOTAL_WORKERS}: Processing {len(worker_symbols)} symbols")

        # Initialize fetcher
        fetcher = TwelveDataFetcher()

        # Fetch with fault tolerance
        results = await fetcher.fetch_batch(worker_symbols)

        # Insert to BigQuery with retry
        if results:
            await insert_to_bigquery(results)

        return {
            'worker_id': WORKER_ID,
            'symbols_processed': len(worker_symbols),
            'records_fetched': len(results),
            'stats': fetcher.stats
        }

    result = asyncio.run(main())
    return result, 200
```

```python
@exponential_backoff_retry(max_retries=3, base_delay=2.0)
async def insert_to_bigquery(records: List[Dict]):
    """Insert records to BigQuery with retry"""
    from google.cloud import bigquery

    client = bigquery.Client()
    table_id = f"{os.getenv('GCP_PROJECT_ID')}.{os.getenv('BIGQUERY_DATASET')}.crypto_daily"

    errors = client.insert_rows_json(table_id, records)

    if errors:
        logger.error(f"BigQuery insert errors: {errors}")
        raise Exception(f"BigQuery insert failed: {errors}")

    logger.info(f"Inserted {len(records)} records to BigQuery")


def get_crypto_symbols() -> List[str]:
    """Get list of crypto symbols to fetch"""
    # Could be from config file, BigQuery, or hardcoded list
    # Example: Top 100 crypto pairs
    return [
        'BTC/USD', 'ETH/USD', 'BNB/USD', 'XRP/USD', 'ADA/USD',
        # ... etc
    ]
```

Cloud Scheduler Integration

```bash
```

```
# Deploy parallel fetchers
for i in {0..9}; do
  gcloud functions deploy daily-crypto-fetcher-$i \
    --gen2 --runtime=python311 \
    --region=us-central1 \
    --memory=2GB --timeout=540s \
    --set-env-vars="WORKER_ID=$i,TOTAL_WORKERS=10,ENVIRONMENT=production"
done

# Create scheduler jobs to trigger all workers
gcloud scheduler jobs create http daily-crypto-trigger \
  --location=us-central1 \
  --schedule="0 0 * * *" \
  --time-zone="America/New_York" \
  --uri="https://us-central1-cryptobot-462709.cloudfunctions.net/daily-crypto-orchestrator" \
  --http-method=POST

# Orchestrator function triggers all 10 workers in parallel
```

Orchestrator Function

python

```python
@functions_framework.http
def daily_crypto_orchestrator(request):
    """
    Orchestrator that triggers all parallel worker functions.
    """
    import aiohttp
    import asyncio

    async def trigger_all_workers():
        tasks = []

        for worker_id in range(TOTAL_WORKERS):
            url = f"https://us-central1-cryptobot-462709.cloudfunctions.net/daily-crypto-fetcher-{worker_id}"
            tasks.append(trigger_worker(url, worker_id))

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

    @exponential_backoff_retry(max_retries=3)
    async def trigger_worker(url: str, worker_id: int):
        async with aiohttp.ClientSession() as session:
            async with session.post(url, timeout=aiohttp.ClientTimeout(total=600)) as response:
                return await response.json()

    results = asyncio.run(trigger_all_workers())

    # Aggregate stats
    total_success = sum(r.get('stats', {}).get('success', 0) for r in results if isinstance(r, dict))
    total_failures = sum(r.get('stats', {}).get('failures', 0) for r in results if isinstance(r, dict))

    return {
        'status': 'completed',
        'workers_triggered': TOTAL_WORKERS,
        'total_success': total_success,
        'total_failures': total_failures,
        'worker_results': results
    }, 200
```

🚨 Alerting & Monitoring

python

```
from collections import defaultdict

class AlertManager:
    """
    Manages alerts for fault tolerance events.
    Integrates with Cloud Monitoring.
    """

    ALERT_THRESHOLDS = {
        'consecutive_failures': int(os.getenv('ALERT_CONSECUTIVE_FAILURES', 3)),
        'error_rate_pct': float(os.getenv('ALERT_ERROR_RATE_PCT', 10.0)),
        'gap_duration_hours': float(os.getenv('ALERT_GAP_HOURS', 4.0)),
    }

    def __init__(self):
        self.error_counts = defaultdict(int)
        self.total_counts = defaultdict(int)

    async def check_and_alert(self, provider: str, success: bool):
        """Track success/failure and alert if thresholds exceeded"""
        self.total_counts[provider] += 1

        if not success:
            self.error_counts[provider] += 1

            # Check consecutive failures
            if self.error_counts[provider] >= self.ALERT_THRESHOLDS['consecutive_failures']:
                await self._send_alert(
                    severity='high',
                    message=f"Provider {provider}: {self.error_counts[provider]} consecutive failures",
                    provider=provider
                )
        else:
            self.error_counts[provider] = 0

        # Check error rate
        if self.total_counts[provider] >= 100:
            error_rate = (self.error_counts[provider] / self.total_counts[provider]) * 100
            if error_rate >= self.ALERT_THRESHOLDS['error_rate_pct']:
                await self._send_alert(
                    severity='medium',
                    message=f"Provider {provider}: {error_rate:.1f}% error rate",
                    provider=provider
```

```
            )

    async def _send_alert(self, severity: str, message: str, provider: str):
        """Send alert to Cloud Monitoring and logging"""
        from google.cloud import logging as cloud_logging

        alert_payload = {
            'severity': severity,
            'message': message,
            'provider': provider,
            'timestamp': datetime.utcnow().isoformat(),
            'environment': os.getenv('ENVIRONMENT')
        }

        # Log to Cloud Logging with severity
        client = cloud_logging.Client()
        logger = client.logger('fault-tolerance-alerts')

        if severity == 'high':
            logger.log_struct(alert_payload, severity='ERROR')
        else:
            logger.log_struct(alert_payload, severity='WARNING')

        # Could also integrate with PagerDuty, Slack, etc.
        logger.info(f"ALERT [{severity}]: {message}")


# Global alert manager
alert_manager = AlertManager()
```

📊 Monitoring Dashboard Metrics

Track these metrics for fault tolerance health:

Metric	Description	Alert Threshold
`api.retry.count`	Total retry attempts per provider	> 10/min
`api.circuit.state`	Circuit breaker state (0=closed, 1=open)	= 1
`api.rate_limit.remaining`	Remaining rate limit capacity	< 10%
`data.gaps.detected`	Number of data gaps found	> 0
`data.validation.failed`	Records failing validation	> 1%
`api.latency.p99`	99th percentile API latency	> 5s
`kraken.nonce.resyncs`	Nonce resync events	> 3/hour
`twelvedata.credits.used`	Daily credit utilization	> 90%

📋 QUICK REFERENCE

Deploy All Components

```bash
# 1. Deploy Cloud Functions
./scripts/deploy_functions.sh

# 2. Create schedulers
./scripts/create_schedulers.sh

# 3. Set up monitoring
./scripts/setup_monitoring.sh
```

Emergency Controls

```bash
```

```
# Pause all data collection
gcloud scheduler jobs pause daily-crypto-trigger --location=us-central1
gcloud scheduler jobs pause hourly-crypto-trigger --location=us-central1
gcloud scheduler jobs pause fivemin-crypto-trigger --location=us-central1

# Resume
gcloud scheduler jobs resume daily-crypto-trigger --location=us-central1
```

Health Check Query

```
sql

SELECT
  DATE(datetime) as date,
  COUNT(DISTINCT symbol) as unique_symbols,
  COUNT(*) as total_records,
  COUNTIF(growth_score IS NOT NULL) as with_growth_score
FROM `cryptobot-462709.crypto_trading_data.crypto_analysis`
WHERE DATE(datetime) >= CURRENT_DATE() - 7
GROUP BY date
ORDER BY date DESC;
```

📡 MAXIMUM API CAPACITY EXPLOITATION

Overview: Multi-Provider Data Strategy

This section defines how to maximize utilization of ALL paid API subscriptions while maintaining full fault tolerance. The goal is to extract maximum value from every API credit.

```
┌─────────────────────────────────────────────────────────────────┐
│                API CAPACITY TARGETS BY PROVIDER                 │
├─────────────────────────────────────────────────────────────────┤
│ TWELVEDATA PRO ($229/mo)                                        │
│   Daily Capacity:    1,152,000 credits                          │
│   Target Usage:      576,000 credits/day (50%)                  │
│   Current Usage:     7,200 credits/day (0.63%)                  │
│   Gap to Close:      80x increase needed                        │
│                                                                 │
│ FRED (FREE)                                                     │
│   Rate Limit:        120 requests/minute                        │
│   Daily Capacity:    172,800 requests (unlimited)               │
│   Target Usage:      10,000 requests/day                        │
│   Data Type:         Macroeconomic indicators                   │
│                                                                 │
│ FINNHUB (FREE/PAID)                                             │
│   Rate Limit:        60 calls/minute (free), 300/min (paid)     │
│   Daily Capacity:    86,400 calls (free tier)                   │
│   Target Usage:      50,000 calls/day                           │
│   Data Type:         News, sentiment, company data              │
│                                                                 │
│ COINMARKETCAP ($79/mo)                                          │
│   Monthly Credits:   10,000 (Standard tier)                     │
│   Daily Target:      333 credits/day                            │
│   Data Type:         Crypto metadata, rankings, global metrics  │
│                                                                 │
│ KRAKEN PRO                                                      │
│   Counter Max:       20 (decays 1/sec for Pro)                  │
│   Effective Rate:    ~60 requests/minute sustained              │
│   Daily Capacity:    86,400 requests                            │
│   Target Usage:      Real-time WebSocket + 20,000 REST/day      │
└─────────────────────────────────────────────────────────────────┘
```

📊 TWELVEDATA MAXIMUM EXPLOITATION

Endpoint Coverage (ALL 35+ Endpoints)

yaml

```
TWELVEDATA_ENDPOINTS:
  # ============================================
  # PRICE DATA (Primary)
  # ============================================
  time_series:
    description: "OHLCV historical data"
    credit_cost: 1 per symbol
    batch_size: 8 symbols
    max_outputsize: 5000
    usage: "20-year backfill for stocks, 10-year for ETFs"

  quote:
    description: "Real-time quote"
    credit_cost: 1 per symbol
    batch_size: 8 symbols
    usage: "Intraday updates"

  price:
    description: "Latest price only"
    credit_cost: 1 per symbol
    batch_size: 8 symbols
    usage: "Lightweight price checks"

  eod:
    description: "End-of-day prices"
    credit_cost: 1 per symbol
    batch_size: 8 symbols
    usage: "Daily close capture"

  # ============================================
  # TECHNICAL INDICATORS (112 Available)
  # ============================================
  technical_indicators:
    description: "All 112 technical indicators"
    credit_cost: 1 per indicator per symbol
    batch_size: 8 symbols
    indicators_per_request: 1
    usage: "Full indicator suite daily"

    indicator_list:
      trend: [sma, ema, wma, dema, tema, kama, t3, vwma, hma, zlema, trima, mama, frama]
      momentum: [rsi, stoch, stochrsi, cci, williams_r, roc, mom, ao, ppo, pvo, trix, uo, cmo]
      volatility: [atr, natr, trange, bbands, keltner, donchian, stddev, variance]
```

```yaml
    volume: [obv, ad, adosc, vwap, mfi, nvi, pvi, emv, vpt, fi]
    overlap: [sar, supertrend, pivots, ichimoku]
    pattern: [cdl_patterns]  # 60+ candlestick patterns

# ================================================
# FUNDAMENTALS (ALL ENDPOINTS - CURRENTLY UNUSED!)
# ================================================
earnings:
  description: "Earnings history & estimates"
  credit_cost: 10 per symbol
  usage: "Quarterly for all 500 stocks"
  data_depth: "20 quarters"

earnings_calendar:
  description: "Upcoming earnings dates"
  credit_cost: 1
  usage: "Daily calendar check"

dividends:
  description: "Dividend history"
  credit_cost: 10 per symbol
  usage: "Full history for dividend stocks"

dividends_calendar:
  description: "Ex-dividend dates"
  credit_cost: 1
  usage: "Daily calendar check"

income_statement:
  description: "Income statement"
  credit_cost: 20 per symbol
  usage: "Quarterly for all stocks"
  periods: 20

balance_sheet:
  description: "Balance sheet"
  credit_cost: 20 per symbol
  usage: "Quarterly for all stocks"
  periods: 20

cash_flow:
  description: "Cash flow statement"
  credit_cost: 20 per symbol
  usage: "Quarterly for all stocks"
```

```
  periods: 20

statistics:
  description: "Key statistics"
  credit_cost: 10 per symbol
  usage: "Weekly refresh"

profile:
  description: "Company profile"
  credit_cost: 1 per symbol
  usage: "Monthly refresh"

analyst_recommendations:
  description: "Analyst ratings"
  credit_cost: 10 per symbol
  usage: "Weekly refresh"

price_target:
  description: "Price targets"
  credit_cost: 10 per symbol
  usage: "Weekly refresh"

institutional_holders:
  description: "Institutional ownership"
  credit_cost: 10 per symbol
  usage: "Quarterly refresh"

insider_transactions:
  description: "Insider trading"
  credit_cost: 10 per symbol
  usage: "Weekly refresh"
```

TwelveData Fault-Tolerant Fetcher

python

```
class TwelveDataMaxCapacityFetcher:
    """
    Maximizes TwelveData API utilization with full fault tolerance.
    Target: 576,000 credits/day (50% of Pro plan capacity)
    """

    DAILY_TARGET = 576000
    BATCH_SIZE = 8
    PARALLEL_WORKERS = 20

    def __init__(self):
        self.circuit = CIRCUIT_BREAKERS['twelvedata']
        self.api_key = os.getenv('TWELVEDATA_API_KEY')
        self.base_url = os.getenv('TWELVEDATA_BASE_URL')
        self.daily_usage = 0
        self.session_start = datetime.utcnow()

        # Endpoint costs
        self.COSTS = {
            'time_series': 1,
            'quote': 1,
            'price': 1,
            'technical_indicators': 1,
            'earnings': 10,
            'dividends': 10,
            'income_statement': 20,
            'balance_sheet': 20,
            'cash_flow': 20,
            'statistics': 10,
            'profile': 1,
            'analyst_recommendations': 10,
            'price_target': 10,
            'institutional_holders': 10,
            'insider_transactions': 10,
        }

    async def execute_daily_collection_plan(self):
        """
        Execute comprehensive daily data collection.
        Allocates credits across all data types.
        """
        plan = self._calculate_daily_plan()
        results = {}
```

```
    # Phase 1: Price Data (150,000 credits)
    results['price_data'] = await self._collect_price_data(plan['price_allocation'])

    # Phase 2: Technical Indicators (150,000 credits)
    results['indicators'] = await self._collect_indicators(plan['indicator_allocation'])

    # Phase 3: Fundamentals (100,000 credits)
    results['fundamentals'] = await self._collect_fundamentals(plan['fundamental_allocation'])

    # Phase 4: Intraday Updates (100,000 credits)
    results['intraday'] = await self._collect_intraday(plan['intraday_allocation'])

    # Phase 5: Historical Backfill (76,000 credits - ongoing)
    results['backfill'] = await self._continue_backfill(plan['backfill_allocation'])

    return results

def _calculate_daily_plan(self) -> dict:
    """Calculate credit allocation for the day"""
    return {
        'price_allocation': 150000,      # Daily OHLCV for 960 symbols × 4 timeframes
        'indicator_allocation': 150000,  # 24 indicators × 960 symbols
        'fundamental_allocation': 100000, # Earnings, financials, etc.
        'intraday_allocation': 100000,   # Hourly + 5-min for top 200
        'backfill_allocation': 76000,    # Historical data catch-up
        'total': 576000
    }

@exponential_backoff_retry(max_retries=5, base_delay=1.0)
async def fetch_batch(self, endpoint: str, symbols: List[str],
                      params: dict = None) -> List[dict]:
    """
    Fetch data for batch of symbols with fault tolerance.
    """
    if not self.circuit.can_execute():
        raise CircuitOpenError("TwelveData circuit breaker open")

    # Calculate cost
    cost = self.COSTS.get(endpoint, 1) * len(symbols)

    # Check daily budget
    if self.daily_usage + cost > self.DAILY_TARGET:
        logger.warning(f"Daily target reached: {self.daily_usage}/{self.DAILY_TARGET}")
```

```
            return []

        # Acquire rate limit
        await rate_limiter.wait_for_capacity('twelvedata', cost=cost)

        try:
            params = params or {}
            params['symbol'] = ','.join(symbols[:self.BATCH_SIZE])
            params['apikey'] = self.api_key

            async with aiohttp.ClientSession() as session:
                url = f"{self.base_url}/{endpoint}"

                async with session.get(url, params=params,
                                       timeout=aiohttp.ClientTimeout(total=60)) as response:

                    if response.status == 429:
                        retry_after = int(response.headers.get('Retry-After', 60))
                        rate_limiter.handle_429('twelvedata', retry_after)
                        raise RetryableHTTPError("Rate limited", 429)

                    response.raise_for_status()
                    data = await response.json()

                    # Check for API errors
                    if data.get('status') == 'error':
                        error_msg = data.get('message', 'Unknown error')
                        if 'rate limit' in error_msg.lower():
                            raise RetryableHTTPError(error_msg, 429)
                        raise Exception(f"TwelveData API error: {error_msg}")

                    self.daily_usage += cost
                    self.circuit.record_success()

                    return self._parse_response(data, symbols, endpoint)

        except Exception as e:
            self.circuit.record_failure()
            raise

    async def fetch_all_fundamentals(self, symbol: str) -> dict:
        """
        Fetch ALL fundamental data for a symbol.
        Cost: ~100 credits per symbol (comprehensive)
```

```
    """
    fundamentals = {}

    endpoints = [
        ('earnings', {'symbol': symbol, 'outputsize': 20}),
        ('dividends', {'symbol': symbol}),
        ('income_statement', {'symbol': symbol, 'period': 'quarterly'}),
        ('balance_sheet', {'symbol': symbol, 'period': 'quarterly'}),
        ('cash_flow', {'symbol': symbol, 'period': 'quarterly'}),
        ('statistics', {'symbol': symbol}),
        ('profile', {'symbol': symbol}),
        ('analyst_recommendations', {'symbol': symbol}),
        ('price_target', {'symbol': symbol}),
        ('institutional_holders', {'symbol': symbol}),
        ('insider_transactions', {'symbol': symbol}),
    ]

    for endpoint, params in endpoints:
        try:
            data = await self.fetch_batch(endpoint, [symbol], params)
            fundamentals[endpoint] = data
            await asyncio.sleep(0.1)  # Small delay between calls
        except Exception as e:
            logger.error(f"Failed to fetch {endpoint} for {symbol}: {e}")
            fundamentals[endpoint] = {'error': str(e)}

    return fundamentals

async def fetch_all_indicators(self, symbol: str, interval: str = '1day') -> dict:
    """
    Fetch ALL 112 technical indicators for a symbol.
    Cost: ~112 credits per symbol
    """
    indicators = {}

    # All available indicators
    INDICATOR_LIST = [
        'sma', 'ema', 'wma', 'dema', 'tema', 'trima', 'kama', 'mama', 't3',
        'rsi', 'stoch', 'stochrsi', 'willr', 'cci', 'roc', 'mom', 'ao', 'cmo',
        'atr', 'natr', 'trange', 'bbands', 'percent_b', 'stddev', 'variance',
        'obv', 'ad', 'adosc', 'vwap', 'mfi', 'adx', 'dx', 'plus_di', 'minus_di',
        'sar', 'aroon', 'aroonosc', 'macd', 'ppo', 'trix', 'ultosc', 'uo',
        'keltner', 'donchian', 'ichimoku', 'pivot_points_hl',
        'supertrend', 'vwma', 'hma', 'zlema',
```

```
]

for indicator in INDICATOR_LIST:
    try:
        params = {
            'symbol': symbol,
            'interval': interval,
            'outputsize': 100,
        }

        # Add indicator-specific parameters
        if indicator in ['sma', 'ema', 'wma', 'rsi', 'atr']:
            params['time_period'] = 14

        data = await self.fetch_batch(f'{indicator}', [symbol], params)
        indicators[indicator] = data

    except Exception as e:
        logger.warning(f"Failed {indicator} for {symbol}: {e}")
        indicators[indicator] = No
```